Universidad Autónoma de Madrid

Escuela Politécnica Superior

Máster en I2-ISIC

TRABAJO DE FIN DE MÁSTER

RECOMENDACIÓN BASADA EN PROCESOS DE DECISIÓN DE MARKOV

Adrián Pertejo Mangas
Tutor: Pablo Castells Azpilicueta

24 de junio de 2020

RECOMENDACIÓN BASADA EN PROCESOS DE DECISIÓN DE MARKOV

Autor: Adrián Pertejo Mangas
Tutor: Pablo Castells Azpilicueta

Departamento de Informática
Escuela Politécnica Superior
Universidad Autónoma de Madrid

24 de junio de 2020

Abstract

Abstract — In recent years, the field of Reinforcement Learning has acquired significant relevance in many areas of study. Through the proposal of an interactive learning paradigm, it seeks to model the idea of natural learning that a living being has in relation to its environment. In the literature, work has begun on the integration of this new area with classic Machine Learning problems, and among them, the recommendation is no exception.

During this work, a study of the field of Reinforcement Learning has been carried out, with the intention of understanding how these algorithms work, and being able to propose a recommendation system based on it. From this study, we move on to the development of a system based on Deep Q-Learning, one of the most promising algorithms in the field, which uses a neural network as an approximator to estimate the value function. Furthermore, using matrix factorization, we extract the latent features of users and objects, and use them to model the agent states, guided by proposals already presented in the literature.

Once the system is developed, we perform experiments in order to first evaluate the effectiveness of different Deep Q-Learning strategies, using a dataset that represents a cold start, where there is little training data. Finally, we compare the performance of the most interesting strategy of the previous step, with other classical recommendation algorithms, based on collaborative filtering: near neighbours, matrix factorization, etc.

Key words — Information Retrieval, Recommendation Systems, Reinforcement Learning, Deep Learning, Matrix Factorization, Collaborative Filtering.

Resumen

Resumen — En los últimos años, el campo del Aprendizaje por Refuerzo (Reinforcement Learning) ha adquirido una relevancia notoria en múltiples áreas de estudio. Mediante la propuesta de un paradigma de aprendizaje interactivo, busca modelar la idea del aprendizaje natural que tiene un ser vivo en relación con su entorno. En la literatura, se ha empezado a trabajar en la integración de este nuevo área con problemas clásicos de Machine Learning, y entre ellos, la recomendación no es una excepción.

Durante este trabajo, se ha realizado un estudio del campo del Reinforcement Learning, con la intención de comprender cómo funcionan estos algoritmos, y poder proponer un sistema de recomendación basado en él. A partir de este estudio, se transita al desarrollo de un sistema basado en Deep Q-Learning, uno de los algoritmos más prometedores del campo, que utiliza una red neuronal como aproximador para estimar la función de valor. Además, usando factorización de matrices, extraemos las características latentes de usuarios y objetos, y las usamos para modelar los estados del agente, guiándonos por propuestas ya presentadas en la literatura.

Una vez desarrollado el sistema, realizamos experimentos con el objetivo de, primero, evaluar la efectividad de diferentes estrategias de Deep Q-Learning, utilizando un conjunto de datos que representa un arranque en frío, donde hay pocos datos de entrenamiento. Por último, comparamos el rendimiento de la estrategia más interesante del paso anterior, con otros algoritmos clásicos de recomendación, basados en filtrado colaborativo: vecinos próximos, factorización de matrices, etc.

Palabras clave — Recuperación de Información, Sistemas de Recomendación, Aprendizaje por Refuerzo, Aprendizaje Profundo, Factorización de Matrices, Filtrado colaborativo.

Índice general

1. Introducción **1**

1.1. Motivación del proyecto 1

1.2. Alcance 2

1.3. Estructura del documento 3

2. Estado del Arte **5**

2.1. Sistemas de Recomendación 5

2.1.1. Matriz de utilidad (ó matriz de puntuaciones) 6

2.1.2. Algoritmos de recomendación 6

2.1.3. Evaluación 10

2.2. Deep Learning 10

2.2.1. Redes Neuronales 11

2.2.2. Entrenando Redes Neuronales 13

2.3. Aprendizaje por refuerzo 17

2.3.1. Políticas 18

2.3.2. Señales de Recompensa 19

2.3.3. Funciones de valor 19

2.3.4. Retornos 19

2.4. Procesos de Decisión de Markov 20

2.4.1. Propiedad de Markov 20

2.4.2. Tipos de MDP 20

2.4.3. Fully Observable MDPs 21

2.4.4. Valores y políticas óptimos 23

2.5. Q Learning 24

2.5.1. Deep Q Learning 24

3. Sistema, diseño y desarrollo **27**

3.1. Simulación del ciclo recursivo de recomendación 27

3.2. Sistema de gestión de ratings 28

3.3. Recomendación basada en RL 29

3.4. Resolviendo el MDP: Programación Dinámica 30

3.4.1. Policy Iteration 30

3.4.2. Value Iteration 31

3.5. Resolviendo el MDP: Q-Learning 33

3.5.1. Deep Q Learning 34
3.6. Implementación de Recomendador basado en Deep Q-Learning 35
3.6.1. Sistema de Estados Latentes 36
3.6.2. Modelo de Q-Network 37
3.6.3. Política utilizada 38
3.6.4. Entorno de estados latentes 39
3.6.5. Tipos de Deep Q-Learning 40
3.6.6. Entrenamiento del recomendador 41

4. Resultados **43**
4.1. Detalles de los conjuntos de datos 43
4.2. Definición de la tarea 45
4.3. Configuración y parámetros 46
4.3.1. Barrido de gamma 47
4.4. Comparativa de técnicas de Deep Q-Learning 49
4.5. Comparativa de algoritmos de recomendación 51

5. Conclusiones y trabajo futuro **55**
5.1. Conclusiones 55
5.2. Trabajo Futuro 56

Bibliografía **59**

Apéndices **63**

A. Código desarrollado **65**
A.1. Definiendo el sistema de gestión de ratings 65
A.2. Definiendo las políticas y el entorno 71
A.3. Definiendo el agente y la Q-Network 74
A.4. Entrenando el agente 77
A.5. Evaluando el agente 83

Índice de tablas

4.1. Mean Reward en test para cada valor de γ. 49
4.2. Mean Reward en test para cada valor de γ. 51

Índice de figuras

2.1. Ejemplo de una matriz de utilidad 6
2.2. Imagen esquemática de una neurona biológica. Imagen extraída de las diapositivas de teoría de la asignatura Neurocomputación, del Grado en Ingeniería Informática, EPS, UAM. 11
2.3. Representación simplificada del modelo McCulloch-Pitts. Imagen extraída de [11]. 12
2.4. Representación de las funciones sigmoidal y ReLU. Imagen extraída del portal Towards Data Science: `https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6` 14
2.5. Ejemplo de una transformación realizada mediante embeddings. Transformamos una variable discreta, libros, a un espacio continuo. En la imagen se observa como se distribuyen los libros por género. Extraída del portal Towards Data Science. 17
2.6. Esquemático de la interacción entre un agente y un entorno en el paradigma del Aprendizaje por Refuerzo. Imagen extraída del libro de Sutton y Barto [18]. 18

3.1. Esquemático de un modelo de Deep Q-Learning. Imagen extraída de [31]. . 35
3.2. Modelo Keras de la Q-Network descrita. 38
3.3. Equilibrio entre explotación y exploración, en función del valor de ϵ. Imagen extraída del portal ResearchGate: `https://www.researchgate.net/figure/Epsilon-greedy-method-At-each-step-a-random-number-is-generated-by-t` `fig2_334741451` 39
3.4. Diagrama general del recomendador basado en Reinforcement Learning. Imagen extraída de [23]. 41

4.1. Distribución de los objetos en ml-100k. 44
4.2. Recompensa acumulada por episodio en la fase de entrenamiento para el barrido de γ. 48
4.3. Recall por episodio en la fase de test para el barrido de γ. 49
4.4. Recompensa acumulada por episodio en la fase de entrenamiento para la comparativa de técnicas de Deep Q-Learning. 50
4.5. Recall por episodio en la fase de test para la comparativa de técnicas de Deep Q-Learning. 51
4.6. Relevancia descubierta para los diferentes algoritmos de recomendación propuestos. 53

1
Introducción

1.1. Motivación del proyecto

El campo del Aprendizaje por Refuerzo (RL, de Reinforcement Learning, sus siglas en inglés) ha sufrido un incremento notable en los últimos años en investigación y en aplicaciones. Surgió con la motivación de modelar computacionalmente el aprendizaje interactivo propio de los seres vivos, y aporta una formulación matemática formal al proceso interactivo entre un agente y un entorno. El primero toma acciones sobre el segundo, alterando su estado. Este paradigma puede ser aplicado a la tarea de recomendación, donde el recomendador actúa como agente, decidiendo qué objetos recomendar y provocando cambios en el entorno (conseguir nueva información sobre un usuario, descubrir ratings relevantes, etc.).

Esta integración no es trivial, y aunque en la literatura ya se han propuesto diferentes aproximaciones, no ha sido estudiada en profundidad. La mayoría de ellas se basan en el uso del Deep Q-Learning, una variación del algoritmo clásico de Reinforcement Learning, Q-Learning. Con este método, se utiliza una red neuronal profunda como aproximador de una función que evalúa como de deseable es cada decisión, la función Q. Sin embargo, a la hora de modelar el sistema, hay multitud de opciones que pueden variar la efectividad del mismo, entre ellas, cómo modelar los estados y las acciones, como implementar las transiciones entre los estados, etc.

Dentro del campo del Reinforcement Learning hay varias formas de modelar este proceso interactivo entre agente y entorno, siendo la más importante y estudiada el modelo del Proceso de Decisión de Markov (MDP, por sus siglas en inglés). Su principal característica es cumplir la Propiedad de Markov, que asegura que el estado actual es independiente de los anteriores. Esta propiedad permite luego aplicar algoritmos más

complejos para resolver el MDP, obteniendo una función que nos indique en cada estado cuál es la mejor acción a tomar: la política.

Por tanto, en este trabajo, se busca investigar exhaustivamente el campo del Aprendizaje por Refuerzo, en concreto, centrándonos en los modelos basados en MDPs, con el objetivo de implementar un recomendador basado en Reinforcement Learning. Esto nos llevará a explorar los principales algoritmos de resolución de MDPs: Policy Iteration, Value Iteration, Q-Learning, entre otros, para explorar cual es el más interesante para la tarea de recomendación, y poder implementar un Sistema de Recomendación basado en él.

1.2. Alcance

El objetivo principal de este Trabajo Fin de Máster es explorar técnicas de Reinforcement Learning, y estudiar su integración con el campo de la recomendación, para poder implementar efectivamente un recomendador basado en RL, y poder comparar su rendimiento contra otros algoritmos de recomendación clásicos. Este objetivo general se puede subdividir en otros objetivos más concretos.

El primero, entender la formalización del paradigma del Reinforcement Learning:

- Realizar un estudio teórico del campo. Entender el modelado del proceso interactivo agente-entorno: estados, acciones, funciones de valor, políticas, recompensas, etc.
- Estudiar el modelo de los Procesos de Decisión de Markov, tanto su planteamiento teórico como los algoritmos para su resolución:
  - Métodos basados en programación dinámica: Policy Evaluation, Policy Iteration, Value Iteration.
  - Métodos basados en diferencias temporales: Q-Learning, Deep Q-Learning.

Después, estudiar la integración con el campo de la recomendación, para poder implementar un recomendador basado en RL:

- Estudiar como integrar ambas áreas, y definir el recomendador basado en RL.
- Implementar el recomendador, utilizando la librería Keras-RL [1] como apoyo durante el proceso.

Por último, realizar un estudio experimental, para comparar el rendimiento tanto de las diferentes estrategias de Reinforcement Learning, como con otros algoritmos de recomendación. Para ello:

- Definir una tarea experimental concisa y representativa del problema.

- Poner a punto un sistema simple de simulación de un entorno de recomendación. Para ello, se parte de unos datos de preferencias de entrada, y se simula el ciclo recursivo en el que se le presentan unas recomendaciones al usuario, y este devuelve un feedback que se incorpora a los datos de entrada. Este feedback se puede implementar de diversas formas, pero por simplicidad, asumimos que el usuario puntúa todas las recomendaciones que le presenta el sistema.
- Implementar las diferentes técnicas de Reinforcement Learning, y los algoritmos simples de recomendación que van a ser usados en los experimentos.
- Ejecutar los experimentos y analizar los resultados obtenidos.

1.3. Estructura del documento

En la Sección 2 se incluye un estudio del estado del arte de las principales áreas involucradas en el proyecto, con el objetivo de contextualizar el trabajo. Estas áreas son Sistemas de Recomendación, Deep Learning, y Reinforcement Learning.

En la Sección 3 se presenta el diseño del sistema desarrollado, incluyendo el proceso de investigación que llevó a diseñar el sistema final.

En la Sección 4 se concretan los experimentos realizados, definiendo las tareas a realizar y las configuraciones escogidas. Se presentan también los resultados de los mismos.

En la Sección 5 cerramos este trabajo con las conclusiones extraídas a raíz del desarrollo realizado y los resultados de los experimentos. Además, planteamos las líneas de un posible trabajo futuro.

Por último, se presenta la bibliografía utilizada en este trabajo.

2
Estado del Arte

En esta Sección se va a contextualizar el trabajo realizando un estudio del arte de las principales áreas involucradas en el mismo: Sistemas de Recomendación en la Sección 2.1, Deep Learning en la Sección 2.2, y, por último, Reinforcement Learning, dividido a su vez en una contextualización general, en la Sección 2.3, seguida de un estudio de Procesos de Decisión de Markov (MDP) en la Sección 2.4, y el estudio del algoritmo central del trabajo, Q-Learning, en la Sección 2.5.

2.1. Sistemas de Recomendación

En los últimos años, con el auge de plataformas a escala masiva como Netflix, Amazon, Spotify, etc. el campo de la recomendación ha alcanzado una importancia considerable. La necesidad de ayudar al usuario a encontrar información, sin que éste necesariamente la haya pedido de forma explícita, es el problema que busca resolver un Sistema de Recomendación. Esta necesidad viene motivada por las cantidades masivas de datos que se manejan actualmente en muchas plataformas, que abruman al usuario cuando éste accede al servicio que le ofrece miles o millones de opciones a elegir. Esta necesidad se traduce a una demanda por recomendaciones personalizadas a cada usuario, teniendo en cuenta sus intereses y preferencias. Es aquí donde entran en juego los Sistemas de Recomendación (RS, de *Recommender System*, en inglés).

Un Sistema de Recomendación se define como un sistema de filtrado de información que extrae lo más relevante de un conjunto masivo de datos, generado por las preferencias, implícitas y/o explícitas, del usuario [2]. Normalmente, estos sistemas manejan información de usuarios, objetos, (ó *items*), y puntuaciones que los usuarios otorgan a los objetos (ó *ratings*). Estos sistemas utilizan algoritmos y tecnologías

muy diversos, pero históricamente se han agrupado en dos categorías [3]: **basados en contenido**, cuando se usa como entrada al algoritmo información sobre los objetos, (normalmente, se usan *features* de los mismos); y **basados en filtrado colaborativo**, cuando se usan los *ratings* conocidos por el sistema para calcular similitudes entre diferentes usuarios y/o entre diferentes objetos.

2.1.1. Matriz de utilidad (ó matriz de puntuaciones)

Como ya hemos comentado, los sistemas de recomendación manejan dos entidades: usuarios y objetos, y las puntuaciones de los primeros a los segundos. Son estas puntuaciones la principal fuente de información de un Sistema de Recomendación. Por lo tanto, debido al gran tamaño que esta información supone en un sistema real, (millones de usuarios, objetos y las puntuaciones entre ellos), se debe utilizar una estructura de datos eficiente. A menudo, en la literatura se maneja un modelo teórico: la Matriz de Utilidad [3] (ó *Utility Matrix*). Se trata de una matriz dispersa, donde las filas representan usuarios, y las columnas objetos, (o por el contrario, filas para objetos y columnas para usuarios), y que almacena las puntuaciones. Como la gran mayoría de estas puntuaciones son desconocidas, todas sus celdas correspondientes estarán vacías, es por ello que hablamos de una matriz. Esta puntuación se puede representar de diversas maneras, aunque típicamente se utiliza un valor numérico, (por ejemplo, un número entre 1 y 5), que representa la puntuación que el usuario le ha dado al objeto, indicando lo mucho o poco que le gusta.

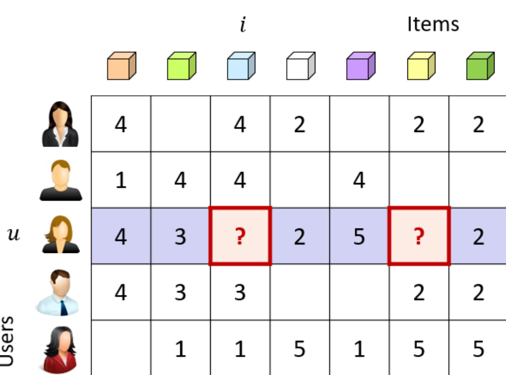

Figura 2.1: Ejemplo de una matriz de utilidad

El problema a resolver por un Sistema de Recomendación se puede formular como generar puntuaciones para aquellas celdas que están vacías. Para ello, se utilizan diversos algoritmos, que son brevemente explicados en la siguiente sección.

2.1.2. Algoritmos de recomendación

Existen infinidad de algoritmos diferentes para resolver la tarea de recomendación. Sin embargo, el funcionamiento genérico de un algoritmo de recomendación se puede modelar

de la siguiente manera:

- Se invoca al algoritmo, con las puntuaciones conocidas por el sistema, un usuario U objetivo y un número máximo de objetos a recomendar, como entradas del mismo.
- El algoritmo asigna a cada objeto conocido por el sistema una puntuación, (ó *score*). Es en el cálculo de esta puntuación donde cada algoritmo se diferencia del resto.
- El algoritmo ordena los objetos de mayor a menor según la puntuación otorgada a cada uno. Se limita la lista (ó *ranking*), con el número máximo de objetos a recomendar.
- Devuelve la lista.

Es interesante revisar la taxonomía de los algoritmos de recomendación, para entender mejor sus diferencias y qué diferentes enfoques existen para extraer recomendaciones. En la literatura [4] [5], se plantea una división de estos algoritmos según su:

- Personalización. Indica con qué grado la recomendación tiene en cuenta los intereses específicos de cada usuario. Un ejemplo de un algoritmo no personalizado es uno basado en la popularidad del objeto, ya que esta no depende de cada usuario. Un algoritmo personalizado podría ser uno basado en la similitud entre usuarios, ya que, obviamente, esas similitudes variarán de un usuario a otro.
- Hibridación. Indica si el algoritmo se compone de una sola estrategia de recomendación, o tiene en cuenta varias de ellas. Un algoritmo simple sería popularidad, por ejemplo, y otro combinado sería uno que tuviera en cuenta tanto la popularidad del objeto como la similitud entre objetos o usuarios.

A parte de estas dos dimensiones, también se dividen los algoritmos según el tipo de sistema en el que son ejecutados:

- Basados en contenido. Estos algoritmos tienen en cuenta información de los objetos, como pueden ser las características del mismo. Se basan en la asunción de que si dos objetos tienen características parecidas, y un usuario U le ha gustado uno de ellos, habrá una alta probabilidad de que le guste también el otro objeto. Un ejemplo es un algoritmo basado en vecinos próximos (kNN, de *k-Nearest-Neighbours*), donde la similitud entre objetos depende de esas características.
- Basados en filtrado colaborativo. Utilizan como entrada las puntuaciones conocidas por el sistema. Un ejemplo puede ser de nuevo un algoritmo de vecinos próximos (kNN), aunque ahora la similitud entre objetos depende sólo de las puntuaciones que han recibido por los usuarios.

2.1.2.1. Algoritmos no personalizados

Estos algoritmos son los más simples, y a menudo son utilizados cuando no se conoce nada del usuario objetivo de la recomendación [4] [5].

- Aleatorios. Se presenta como el algoritmo más sencillo. Recomendar objetos aleatoriamente es una de las fórmulas que menos acierto puede conseguir. Sin embargo, asegura un acierto de media del 50 %, y puede ser útil como punto de partida.
- Basados en popularidad. Estos algoritmos recomiendan los objetos que mayor “popularidad” tienen. Esta popularidad puede ser calculada de diversas maneras: número de usuarios que han puntuado el objeto, puntuación promedio del objeto, etc.

2.1.2.2. Algoritmos personalizados

Estos algoritmos generan recomendaciones personalizadas a cada usuario concreto. Se cuentan por centenares, pero se van a nombrar solo las dos familias más importantes en sistemas basados en filtrado colaborativo, ya que los basados en contenido se computan de manera análoga, intercambiando similitudes basadas en puntuaciones, por similitudes basadas en contenido.

2.1.2.2.1 Vecinos próximos (kNN)

Los algoritmos de esta familia [6] son muy populares, fáciles de entender y además son muy utilizados. Pueden ser orientados a usuarios o a objetos. Generan un vecindario con los N vecinos más próximos al objetivo es decir, que tienen una mayor similitud con el objetivo, ya sea un usuario o un objeto. Estos modelos calculan la predicción de la puntuación de la siguiente forma:

- Basado en usuario.

$$\hat{r}(u,i) = C \sum_{\substack{v \epsilon N_k(u) \\ r(v,i) \neq 0}} sim(u,v) r(v,i)$$

  Donde $N_k(u)$ es el vecindario de u, $\hat{r}(u,i)$ es la predicción de la puntuación del objeto i por el usuario u, y $r(v,i)$ es la puntuación conocida por el sistema que el usuario v ha dado al objeto i.

  C es una constante usada para normalizar. Si se desea que la puntuación predicha esté en el rango $[0,1]$, se puede usar $C = 1/(\sum_{\substack{v \epsilon N_k(u) \\ r(v,i) \neq 0}} |sim(u,v)|)$.

- Basado en objeto.

$$\hat{r}(u,i) = C \sum_{r(u,j)\neq 0} sim(i,j)r(u,j)$$

donde $C = 1/(\sum_{r(u,j)\neq 0} |sim(i,j)|)$.

Para calcular la similitud entre usuarios y objetos, se han presentado en la literatura varias alternativas:

- Similitud coseno.

$$sim(u,v) = \sum_{\substack{i:r(u,i)\neq 0 \\ r(v,i)\neq 0}} r(u,i)r(v,i) / \sqrt{\sum_{i:r(u,i)\neq 0} r(u,i)^2 \sum_{i:r(v,i)\neq 0} r(v,i)^2}$$

- Correlación de Pearson.

$$sim(u,v) = \frac{\sum\limits_{i\epsilon I_{u_v}} (r_{u_i} - \bar{r}_u)(r_{v_i} - \bar{r}_v)}{\sqrt{\sum\limits_{i\epsilon I_{u_v}} (r_{u_i} - \bar{r}_u)^2 \sum\limits_{i\epsilon I_{u_v}} (r_{v_i} - \bar{r}_v)^2}}$$

2.1.2.2.2 Modelos de factores latentes

La principal alternativa a los modelos de vecinos próximos son los basados en factores latentes [4] [5]. Un caso particular muy popular es la factorización de matrices. Este método busca, a partir de la matriz de utilidad del sistema, descubrir características latentes, llamadas factores, que están en concordancia con las puntuaciones obtenidas. Cada componente del vector indica el peso de ese factor latente en un usuario u objeto. Estas aproximaciones están dando muy buenos resultados en los artículos que se publican en los últimos años.

2.1.2.3. Problema de arranque en frío

En los sistemas basados en filtrado colaborativo, surge un problema, conocido como arranque en frío [7], (ó *cold-start problem*), producido cuando se necesitan puntuaciones de un objeto que aún no ha sido puntuado por nadie. Es por ello, que a menudo se combinan sistemas de filtrado colaborativo con otros basados en contenido, para poder solucionar este problema, ya que los basados en contenido pueden detectar similitudes sin necesidad de que los usuarios hayan realizado puntuaciones previas. En [7] se proporcionan estadísticas sobre distintas configuraciones, combinando filtrado colaborativo y contenido, en el contexto de este problema, pero al no ser éste el objetivo de este trabajo, no se va a profundizar más en él.

2.1.3. Evaluación

En la literatura se han propuesto multitud de métricas para evaluar un Sistema de Recomendación. Una de las más importantes el Error Medio Absoluto (ó *Mean Absolute Error (MAE))* [2]:

$$MAE = \frac{1}{N}\sum_{u,i}|\hat{r}(u,i) - r(u,i)|$$

donde $\hat{r}(u,i)$ es la puntuación predicha por el sistema del usuario u al objeto i, $r(u,i)$ es la puntuación real y N es el número total de puntuaciones.

Sin embargo, más recientemente se ha empezado a recomendar el uso de otro tipo de métricas, conocidas como "métricas de apoyo a la precisión de la decisión"[2], siendo las más utilizadas la Precisión y el Recall. Se suele usar la notación P@K, para indicar la Precisión de una recomendación de K objetos, o R@K, análogamente para el Recall:

$$P@K = \frac{|\text{Objetos relevantes en top K}|}{K}$$

$$R@K = \frac{|\text{Objetos relevantes en top K}|}{|\text{Objetos relevantes}|}$$

Aquí se incluye un nuevo concepto: la relevancia [8]. En el campo de la recuperación de información, se distingue entre datos relevantes y no relevantes. Típicamente, un dato relevante es aquel que satisface al usuario. En concreto, en lo referente a una recomendación, es relevante si ésta le gusta al usuario. Normalmente se define un umbral de relevancia, y se dice que toda recomendación con una puntuación mayor o igual a ese umbral es relevante, y menor es no relevante.

2.2. Deep Learning

El Aprendizaje Profundo, ó *Deep Learning* en inglés, es una rama del Aprendizaje Automático, que modela abstracciones complejas de datos, que requieren de transformaciones no lineales, por lo que suelen utilizarse redes neuronales profundas.

Es por tanto importante poner en contexto el estado del arte de los modelos de redes neuronales, y los algoritmos usados en Deep Learning para entrenarlas eficazmente.

2.2.1. Redes Neuronales

Los algoritmos de redes neuronales son algoritmos "bioinspirados". Es decir, surgieron cómo un intento de imitar computacionalmente un modelo natural y existente como es la interacción de las neuronas biológicas de los sistemas nerviosos de los mamíferos. Es por ello, que vamos a definir primero las neuronas biológicas, para poder hacer una analogía más adelante, y poder identificar qué partes de la neurona artificial corresponden con las de la neurona biológica, y entender así mejor el modelo computacional.

2.2.1.1. Neurona biológica

El modelo de neurona biológica que se enseña hoy en día fue originado por Ramón y Cajal, en 1906, con la publicación de su "Doctrina de la Neurona". La teoría original no está disponible en repositorios públicos, pero en [9] está disponible un artículo que resume el original. Ramón y Cajal instauró el estándar actual, en el que la neurona biológica se considera la unidad básica del sistema nervioso. En la figura 2.2 se puede visualizar la concepción actual de una neurona.

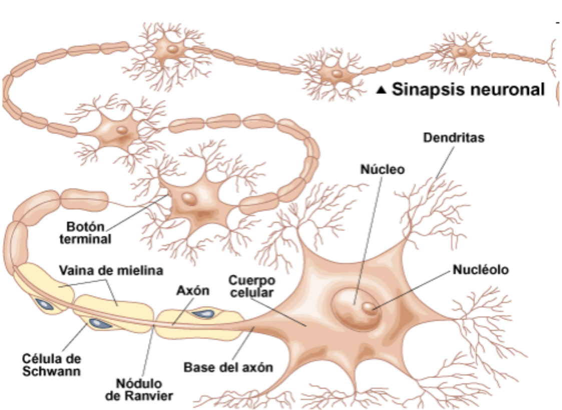

Figura 2.2: Imagen esquemática de una neurona biológica. Imagen extraída de las diapositivas de teoría de la asignatura Neurocomputación, del Grado en Ingeniería Informática, EPS, UAM.

La función principal de las neuronas es recibir, procesar y transmitir información a través de impulsos eléctricos, que generan gracias a la excitabilidad de sus membranas. Se comunican con otras neuronas a través de sinapsis, conformando así una red neuronal biológica.

El proceso de sinapsis es el proceso de conexión y comunicación que utilizan las neuronas para relacionarse entre ellas. Una neurona, llamada célula emisora, genera una descarga química, y segrega neurotransmisores que alcanzan el axón de la neurona receptora. A través de estos neurotransmisores, la neurona receptora se excitará o inhibirá, dependiendo de la situación.

En cuanto a la morfología de una neurona, hay varios elementos que conviene definir.

El primero de ellos son las dendritas. Las dendritas son ramificaciones que salen del cuerpo celular de la neurona, y que se encargan de procesar los neurotransmisores recibidos en una sinapsis. Este cuerpo celular del que salen las dendritas, recibe el nombre de soma. Y a partir de este soma, en dirección contraria a las dendritas, se encuentra una prolongación llamada Axón, que conduce los impulsos desde el soma hacia otras neuronas.

2.2.1.2. Neurona artificial

En 1943, Warren McCulloch y Walter Pitts diseñaron lo que se considera el primer modelo de una red neuronal artificial, publicado en [10]. En su modelo, se pueden identificar varios elementos biológicos. Según este modelo, una red neuronal artificial se compone de:

- Un vector de entrada, X, donde cada entrada será $x_i \in X$.
- Un vector de pesos, W, donde cada peso será $w_i \in W$.
- Una función de activación, a, cuya entrada será una agregación, g, de las entradas.
- Una salida, f, que se computa con la función de activación, $f = a(g(x))$.

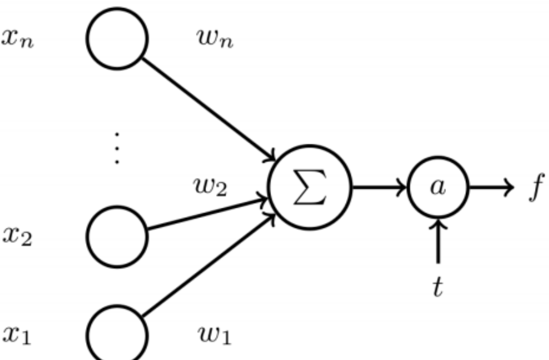

Figura 2.3: Representación simplificada del modelo McCulloch-Pitts. Imagen extraída de [11].

El modelo McCulloch-Pitts se muestra en la figura 2.3. Las entradas, representadas en el vector X, realizan la misma función que las dendritas. Estas se encargan de recibir el impulso de otra neurona, que viaja a través del soma hacia el axón, que lo lleva a su vez hasta las dendritas de la siguiente neurona. Aquí, la salida f simula ese impulso que viaja a través del axón. La función de activación, a, puede ser lineal o no lineal. Un ejemplo muy sencillo, sería que la función de activación se activara si se superase un umbral, comúnmente conocido como el umbral de activación:

$$f = \begin{cases} 1, & \text{si } \sum_{i=1}^{n} x_i w_i > t \\ 0, & \text{en cualquier otro caso} \end{cases}$$

2.2.2. Entrenando Redes Neuronales

Las redes neuronales son aproximadores muy potentes, pero necesitan de un entrenamiento previo para poder aprender. Resuelven problemas de clasificación y regresión, y por tanto, necesitan que se les presente una serie de casos ya clasificados, para aprender a clasificar nuevos ejemplos. El aprendizaje de una red neuronal se concentra en los pesos de sus enlaces, y por tanto, cualquier algoritmo de entrenamiento de una red neuronal modificará esos pesos con el objetivo de minimizar una función de error determinada. Una técnica muy utilizada es la del descenso por gradiente. Generalizando, la actualización de pesos puede definirse de la siguiente forma:

$$W_t = W_{t-1} - \alpha \overrightarrow{\nabla E}$$

donde α es la tasa de aprendizaje, y $\overrightarrow{\nabla E}$ es el gradiente del error.

Hay múltiples modelos de redes neuronales, y múltiples algoritmos de entrenamiento: Regla Delta es uno muy sencillo, utilizado para entrenar redes de una sola capa, y retropropagación, su generalización para redes profundas, entre otros. Ya que el objetivo es poner en contexto el problema de Deep Learning, y este suele implicar resolver problemas no lineales, vamos a definir en profundidad el algoritmo de retropropagación.

2.2.2.1. Algoritmo de retropropagación

Las redes neuronales de una sola capa resuelven problemas lineales, y su entrenamiento es muy sencillo. Sin embargo, si necesitamos resolver un problema no lineal, necesitamos hacer uso de una red neuronal profunda. Estas redes neuronales cuentan con un algoritmo de entrenamiento muy potente: el algoritmo de retropropagación [12], ó *backpropagation* en inglés. Los problemas de Deep Learning suelen implicar relaciones no lineales. Es por esto que el algoritmo de retropropagación es un pilar básico en el área.

Este algoritmo utiliza el descenso por gradiente introducido en el apartado anterior para minimizar el error cuadrático total en la salida. Para ello, el algoritmo se divide en 3 partes. Primero, el patrón de entrada se propaga hacia delante en la red (fase de feedforward). Después, se calcula el error asociado, viendo la diferencia entre la salida obtenida y, y la salida esperada t. Tras esto, se retropropaga este error, desde las últimas capas hacia las primeras. Y por último, se ajustan los pesos de la red. Estos pasos se repiten mientras el error cuadrático medio no haya alcanzado un mínimo admisible.

Aunque el entrenamiento es muy costoso, en la fase de explotación, tan solo es necesario propagar hacia delante el patrón de entrada, por lo que, aunque el entrenamiento de un perceptrón multicapa es muy lento, se compensa con la rapidez con la que clasifica nuevos ejemplos en la fase de explotación.

2.2.2.2. Activación ReLU

En las redes neuronales, la función de activación es uno de los elementos más determinantes, ya que determina cómo se transforma la suma ponderada de las entradas para obtener la salida de cada neurona. Cómo hemos visto, el mecanismo que usa el algoritmo de backpropagation para minimizar el error es el descenso por gradiente. Por tanto, nos interesa que la función de activación sea fácilmente derivable, para que computacionalmente sea rápido calcular ese gradiente.

Una función de activación más intuitiva que la qué vamos a presentar ahora era la función sigmoidal. Esta función se planteó como un buen candidato a función de activación, debido a que cumplía ciertos requisitos: sus posibles valores están en el rango $(0, 1)$, es monótona, pues siempre crece, y es diferenciable en todos sus puntos. Sin embargo, tanto con esta función, como con la tangente hiperbólica (otra función muy usada previamente), apareció el problema de los *Vanishing Gradients*, donde en determinados puntos, el gradiente se va "desvaneciendo" hacia valores muy pequeños, que impiden actualizar correctamente los pesos. Es por ello, que en el dominio del Deep Learning empezó a utilizarse la función ReLU [13], (*Rectified Linear Unit*). En la Figura 2.4 puede observarse las definiciones gráficas de ambas funciones.

La función de activación ReLU otorga valores nulos a entradas negativas, e iguala la salida a la entrada en el caso de entradas positivas. Con esta función, ya no tenemos el problema del desvanecimiento de los gradientes. Además, derivar esta función es mucho más sencillo que derivar la sigmoide, por lo que se acelera el aprendizaje de la red. En concreto, la definición de la derivada de la función ReLU es la siguiente:

$$\frac{\partial R}{\partial z} = \begin{cases} 1, & \text{si } z \geq 0 \\ 0, & \text{si } z < 0 \end{cases}$$

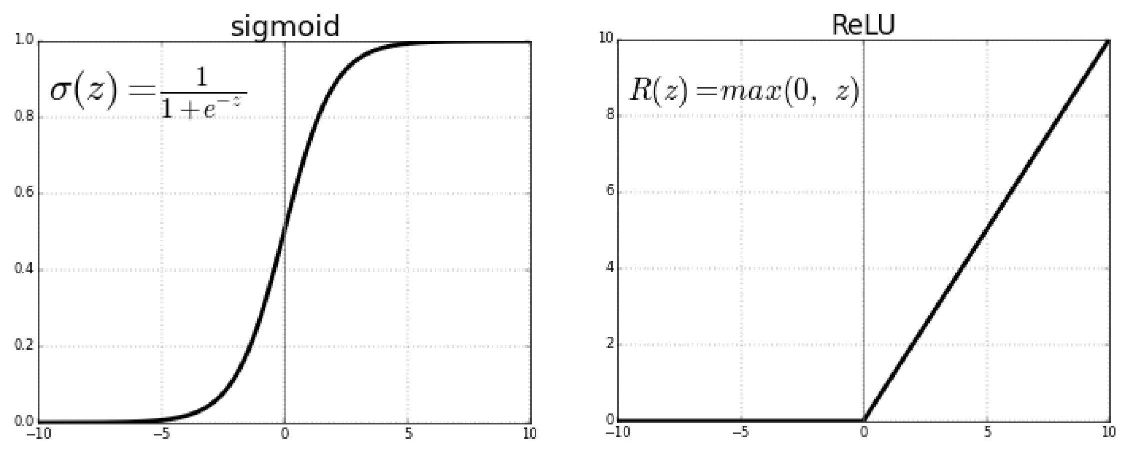

Figura 2.4: Representación de las funciones sigmoidal y ReLU. Imagen extraída del portal Towards Data Science: `https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6`

2.2.2.3. Inicialización

A la hora de entrenar redes profundas, es muy importante escoger con cuidado cómo inicializamos los pesos de la red. Esta inicialización es el punto de partida hacia la deseada convergencia, y según qué punto escojamos, el entrenamiento puede acelerarse mucho, o volverse muy lento, llegando a veces a no converger.

Inicializar los pesos aleatoriamente es la primera opción que surge, aunque no es buena idea por varios motivos. Primero, si los valores que le damos a los pesos son muy grandes, o muy pequeños, corremos el riesgo de que el gradiente se "desvanezca", ya que al aplicar una función de activación, como una sigmoide, obtendremos un valor proximo al 1, donde la dirección del gradiente (la pendiente) cambia muy lentamente y el entrenamiento se hace muy lento. Lo mismo ocurre con valores muy pequeños, donde la función mapeará a valores próximos al 0.

Existen varios métodos que, teniendo en cuenta los problemas mencionados, buscan aportar una inicialización que acelere la convergencia. Hay 2 métodos destacados: el primero, la inicialización de He, propuesto en [14], y el segundo, la inicialización de Xavier Glorot, propuesto en [15]. Ambos métodos son muy parecidos, y ya que el método de Xavier Glorot está más extendido, es el que vamos a definir en este trabajo.

El objetivo del método de Xavier Glorot es mantener la varianza de los pesos entre diferentes capas de la red. Es decir, queremos que la señal que va pasando a través de la red, no alcance un valor demasiado alto, (típicamente, nos referimos a esto cómo que la señal explote), ni que se "desvanezca" hacia el cero. Para ello, inicializamos los pesos escogiéndolos de una distribución, cuya media μ y varianza $Var(w_i)$ son las siguientes:

$$\mu = 0$$
$$Var(w_i) = \frac{1}{n_{in} + n_{out}},$$

dónde n_{in} es el número de neuronas de entrada, ó de la capa anterior; y n_{out} es el número de neuronas de salida, ó de la capa siguiente. Utilizando este método, conseguimos que la varianza se mantenga a través de los pesos de las diferentes capas, y evitamos el problema de que la señal explote o se desvanezca.

2.2.2.4. Regularización

En los problemas de Aprendizaje Automático, surge siempre un fenómeno conocido como el sobreajuste, ó *overfitting*. Se produce cuando el aproximador, en este caso, nuestra red neuronal, aprende demasiado bien los datos de entrada, perdiendo así toda capacidad para generalizar y clasificar bien nuevos ejemplos. En estos casos, el error en entrenamiento es muy bajo, pero el error en test es altísimo. Típicamente, existen 2 formas de resolver este problema: obtener más datos, o usar técnicas de regularización. A menudo la primera opción es imposible, debido al alto coste que puede suponer recolectar nuevos datos. Es por

ello, que es importante entender algunas técnicas de regularización. Vamos a centrarnos en 2 técnicas muy extendidas, el uso de regularización L2, y el uso de dropouts en las redes neuronales.

La regularización L2 [16] añade un nuevo término a la función de coste de la red, que penalizará a los pesos que crezcan demasiado. Por tanto, la función de coste es ahora:

$$C_{L2} = Coste + \lambda ||W||^2,$$

dónde $||W||$ es la norma de la matriz de pesos, y λ es el factor de regularización. Es este factor el que podemos ajustar según el efecto que deseemos conseguir. Los pesos grandes serán más penalizados cuanto más grande hagamos este factor, y lo serán menos si decrece.

Además de este método, existe la opción de utilizar *dropouts* [17]. Con este método, establecemos una probabilidad de que cada nodo de la red sea descartado. Lo que nosotros ajustamos es el umbral de *dropout*. Así, si nuestro umbral es 0,6, hay un 40 % de probabilidades de que un nodo sea descartado. Aunque parezca que no tiene sentido deshacerse de información relevante de la red descartando algunos nodos, este método ha sido probado y se ha demostrado que mejora el rendimiento de las redes neuronales. El razonamiento detrás de este método de regularización, es que la red, al no poder confiar en que un nodo no vaya a desaparecer, no asigna pesos demasiado grandes a ninguna de sus conexiones, porque puede perderlas en cualquier momento. Así, mantiene los pesos bien distribuidos, sin que ninguno sea más importante que el resto, regularizando el modelo de la red.

2.2.2.5. Capas de Embedding

Un embedding es un mapeado de un espacio discreto a uno continuo. Una capa de embedding, por tanto, se encarga de aprender una representación vectorial continua de variables discretas. Además, al realizarse de manera supervisada, en este espacio continuo, los elementos similares discretamente estarán cerca a su vez en el espacio continuo. Esto es una gran ventaja, ya que representaciones más tradicionales, como puede ser la codificación One Hot, tienen el problema de no respetar estas similitudes entre elementos al realizar la transformación. En concreto, en la codificación One Hot, si aplicamos la similitud coseno, siempre obtenemos una similitud nula entre cada par de vectores One Hot.

Estas capas de embedding tienen su origen en el campo del procesamiento de lenguaje natural, donde se usa para transformar el espacio discreto de las palabras a uno continuo. En la figura 2.5 podemos observar un ejemplo de transformación realizada mediante embeddings.

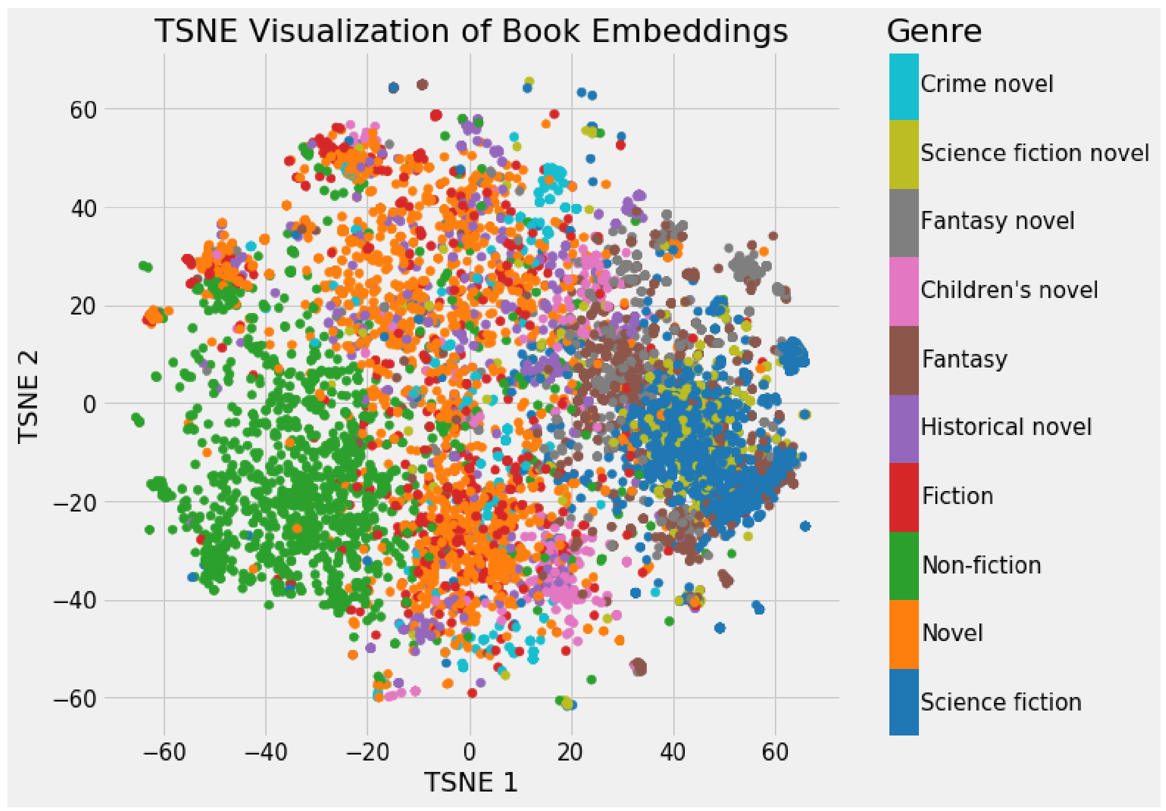

Figura 2.5: Ejemplo de una transformación realizada mediante embeddings. Transformamos una variable discreta, libros, a un espacio continuo. En la imagen se observa como se distribuyen los libros por género. Extraída del portal Towards Data Science.

2.3. Aprendizaje por refuerzo

Cuando un niño se encuentra en las fases tempranas de su vida, intuitivamente aprende siguiendo un proceso natural de interacción con el entorno que lo rodea. Cuando toca algo que quema, el entorno le está mandando información a través de su tacto, que interpreta que tocar algo que quema duele, y por tanto, es peligroso. Este modelo de aprendizaje no es exclusivo a los humanos, está presente en toda la naturaleza. Los animales aprenden ciertos comportamientos a través de esa interacción con lo que les rodea, sin que tengan ningún profesor que les diga exactamente qué deben hacer. El paradigma del Aprendizaje por Refuerzo, ó *Reinforcement Learning, RL* en inglés, surge con la intención de aproximar computacionalmente esta idea tan natural del aprendizaje a través de una interacción con el entorno, que está presente en nuestras vidas desde que nacemos. Este campo conforma una alternativa clara a los algoritmos clásicos de Machine Learning, típicamente clasificados como algoritmos supervisados (clasificación) o no supervisados (clustering).

La idea central del Aprendizaje por Refuerzo [18], es modelar la interacción de un agente con un entorno, a través del tiempo. Este agente puede tomar acciones, que producen cambios en el entorno, y observará una recompensa, que guía su aprendizaje. Estos cambios producidos en el entorno, motivan que se utilicen estados como modelo del estado actual del entorno. En cada paso del sistema, t, el agente toma una acción, observa

el estado siguiente, y extrae una recompensa, que depende del entorno. Esta interacción se puede visualizar en la Figura 2.6.

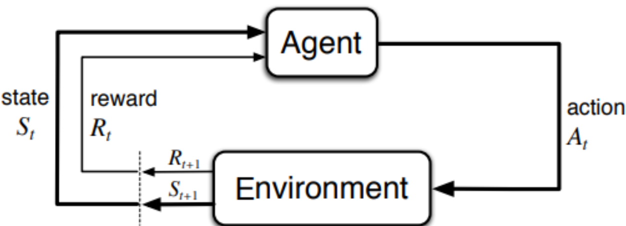

Figura 2.6: Esquemático de la interacción entre un agente y un entorno en el paradigma del Aprendizaje por Refuerzo. Imagen extraída del libro de Sutton y Barto [18].

En un sistema de Aprendizaje por Refuerzo, se suele definir el conjunto de estados posibles como S, siendo uno de esos estados $s \in S$, el de acciones posibles como A, siendo una de esas acciones $a \in A$, y el de recompensas posibles como R, siendo una de esas recompensas $r \in R$.

Una vez definida la idea básica, es importante definir otros elementos que también conforman un sistema de Aprendizaje por Refuerzo. En concreto, los elementos básicos que vamos a introducir son: la política, la señal de recompensa, la función de valor y el concepto de retornos.

2.3.1. Políticas

Una política es una función, que define qué acción tomar dado el estado actual. Es decir, define el comportamiento del agente en un determinado momento. Esta decisión de qué acción tomar, se puede modelar con políticas estáticas, o estocásticas.

En una política estática, dado un estado s, solo hay una acción posible a ser elegida, con probabilidad 1. Es decir, la política estática es una función $f : S \rightarrow A$, e indica directamente qué acción tomar en cada momento.

Por otro lado, una política estocástica es una función, que define una distribución de probabilidades sobre las acciones ($a \in A$) para cada estado ($s \in S$). Es decir, una política π es una función $f : A, S \rightarrow \pi(a|s)$, siendo esto último la probabilidad de tomar tal acción dado el estado actual.

La política es el núcleo de un sistema de Aprendizaje por Refuerzo, en el sentido de que define completamente el aprendizaje y el comportamiento del agente. Es decir, es en la política donde se almacena el aprendizaje alcanzado por el agente, y el objetivo del problema es aprender una política óptima. Se profundizará en este tema en la Sección de Desarrollo.

2.3.2. Señales de Recompensa

A lo largo de los pasos de un sistema de Aprendizaje por Refuerzo, el entorno manda al agente la recompensa, un número, que se puede definir como función del estado: $f : S \rightarrow \mathbb{R}$. Maximizar esta recompensa a largo plazo es el objetivo de un agente, y es por tanto una heurística que puede utilizarse como métrica de calidad en un instante de tiempo t. Esta recompensa solo depende del estado actual del entorno, y por tanto, la única manera en la que el agente puede influir en ella es mediante las acciones que toma. Estas acciones provocarán un cambio de estado, que a su vez, recibirá una recompensa diferente por parte del entorno.

2.3.3. Funciones de valor

Las recompensas del entorno son inmediatas, se otorgan a un estado concreto del entorno, el estado actual. Sin embargo, existe una variante análoga a la recompensa, pero esta vez computada teniendo en cuenta el largo plazo. La función que otorga un valor a cada estado, indicando cómo de deseable es estar en ese estado a largo plazo, es la función de valor, que puede definirse cómo $f : S \rightarrow \mathbb{R}$. En otras palabras, el valor de un estado es la suma total de recompensas que un agente espera acumular, partiendo desde ese estado. Aunque recompensas y valores son parecidos, en el sentido de qué ambos evalúan como de deseable es encontrarse en un cierto estado, a la hora de tomar decisiones, será la función de valor la qué se tendrá en cuenta, ya que es la que nos ayuda a maximizar las recompensas que obtenemos durante todo el proceso iterativo del Aprendizaje por Refuerzo.

Existen dos tipos de funciones de valor, la que acabamos de describir, es la función de valor del estado. Dado un estado, la función nos indica como de bueno es para el agente, encontrarse en ese estado. Sin embargo, existe otra variante, la función de valor estado-acción. En este caso, dado un estado, la función de valor nos indica cómo de bueno es para el agente tomar una acción. Esta función puede definirse en el espacio $f : S, A \rightarrow \mathbb{R}$.

2.3.4. Retornos

En el apartado anterior hemos introducido sin mencionarlo el concepto de retorno. Hemos visto que el valor de un estado está relacionado con la suma de recompensas que obtendremos de media si partimos desde ese estado. Por tanto, podemos definir formalmente, el retorno esperado como:

$$G_t = R_{t+1} + R_{t+2} + R_{t+3} + ... + R_T,$$

donde T es el paso final. Esta definición sin embargo tiene un problema, ya que cuando la tarea es infinita, es decir, no tiene un paso final, el retorno esperado tiende a ∞. Es por ello que necesitamos introducir la idea del descuento:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... + \gamma^{T-1} R_T = \sum_{i=0}^{\infty} \gamma^i R_{t+i+1},$$

donde γ es el factor de descuento, $0 \leq \gamma \leq 1$. Este factor de descuento nos sirve para regular cómo de importantes son las recompensas inmediatas, en comparación de las recompensas futuras. A medida que γ se aproxima a 1, las recompensas futuras tienen más importancia. Si $\gamma = 0$, tan solo importa la recompensa inmediata. Cabe destacar además, que si $\gamma = 1$, estaríamos en el caso anterior, y en una secuencia infinita, el retorno sería ∞.

2.4. Procesos de Decisión de Markov

Cuando una tarea de Aprendizaje por Refuerzo, cuenta con estados que cumplen la propiedad de Markov, recibe el nombre de Proceso de Decisión de Markov (ó MDP, de *Markov Decission Process*).

2.4.1. Propiedad de Markov

Un estado, $S_t \in S$, es un estado de Markov, si cumple la siguiente propiedad:

$$P[S_{t+1}|S_1, ..., S_t] = P[S_{t+1}|S_t]$$

Es decir, el estado es de Markov si contiene toda la información útil que el agente había obtenido hasta ese momento, o dicho de otra manera, que los estados que se han recorrido anteriormente pueden ser deshechados, ya que su información está incluida en el último de ellos.

Otra forma de verlo, es que el futuro es independiente del pasado, dado el presente. Análogamente, conocido el estado actual, los estados anteriores pueden ser descartados ya que no aportan nueva información relevante que no esté ya contenida en el estado actual.

2.4.2. Tipos de MDP

Si los espacios de acciones y de estados del agente son finitos, hablamos entonces de un MDP finito. Es importante definir que son espacios finitos, para poder hablar en los siguientes apartados de probabilidades, y no de funciones de densidad. En el resto del trabajo, asumiremos que hablamos de MDPs finitos, por conveniencia.

A su vez, existen entornos plenamente observables, o parcialmente observables. Cuando el entorno actualiza su estado, y el agente puede observar ese cambio con plenitud, se define

formalmente el proceso como un Proceso de Decisión de Markov Plenamente Observable (ó FOMDP, de *Fully Observable MDP*).

Sin embargo, hay otros entornos que son parcialmente observables, en los que el agente no tiene acceso total al estado del entorno, y no puede observar exactamente cómo cambia. En estos casos, el agente tiene que construir un modelo para definir su propio espacio de estados, a partir de lo que pueda observar del entorno. Se define formalmente este proceso como un Proceso de Decisión de Markov Parcialmente Observable (ó POMDP, de *Partially Observable MDP*). En este trabajo se ha trabajado con la primera opción, así que conceptos y acercamientos específicos de los POMDPs no van a ser tratados en los siguientes apartados.

2.4.3. Fully Observable MDPs

Podemos definir un FOMDP como una tupla (S, A, P, R, γ) [18], donde S es el espacio de estados, A es el espacio de acciones, P es la función de transición, R es la función de recompensa, y γ es el factor de descuento.

La función de transición P define las transiciones entre cada par de estados, dada la acción tomada en ese momento:

$$p(s'|s,a) = Pr\{S_{t+1} = s'|S_t = s, A_t = a\}$$

La función de recompensa R define las recompensas esperadas para cada par estado-acción:

$$r(s,a) = \mathbb{E}[R_{t+1}|S_t = s, A_t = a]$$

El factor de descuento γ, como se introdujo en el Apartado 2.2.4, regula la importancia de las recompensas inmediatas, frente a las futuras.

Estos 5 elementos definen el MDP, pero resolver la tarea de Aprendizaje por Refuerzo suele implicar estimar funciones de valor. Recordemos que, una función de valor es una función que estima cómo de bueno es, ó estar en un estado, ó tomar una acción, dado el estado actual. Además, también hemos introducido el concepto de política. Una política π es una función que asigna a cada par estado-acción, una probabilidad, $\pi(a|s)$. Por tanto, podemos definir, para un FOMDP, el valor de un estado cómo

$$v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \middle| S_t = s\right]$$

$$v_\pi(S_T) = 0,$$

dónde S_T es el estado terminal, si hay alguno, y $\mathbb{E}_\pi$ es la esperanza del retorno, dado que el agente sigue la política π.

Además, podemos definir el valor de una acción, como vimos en el Apartado 2.2.4. Llamamos función de valor acción-estado [18], de una acción a, en un estado s, bajo una política π, a la esperanza del retorno empezando desde el estado s, tomando la acción a, y siguiendo la política π:

$$q_\pi(s,a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty}\gamma^k R_{t+k+1}\middle|S_t = s, A_t = a\right]$$

Estas funciones de valor se pueden desarrollar recursivamente, desarrollando la expresión del retorno esperado. Con esta transformación, llegamos a obtener lo que puede que sea la ecuación más importante de un MDP, la Ecuación de Bellman.

2.4.3.1. Ecuación de Bellman

Si cogemos la expresión inicial de la función de valor, y la desarrollamos [18], obtenemos una relación entre el valor de un estado, y el valor de los estados que le suceden:

$$\begin{aligned}
v_\pi(s) &= \mathbb{E}_\pi[G_t|S_t = s] \\
&= \mathbb{E}_\pi\left[\sum_{k=0}^{\infty}\gamma^k R_{t+k+1}\middle|S_t = s\right] \\
&= \mathbb{E}_\pi\left[R_{t+1} + \gamma\sum_{k=0}^{\infty}\gamma^k R_{t+k+2}\middle|S_t = s\right] \\
&= \sum_a \pi(a|s)\sum_{s'}\sum_r p(s',r|s,a)\left[r + \gamma\mathbb{E}_\pi\left[\sum_{k=0}^{\infty}\gamma^k R_{t+k+2}\middle|S_{t+1} = s'\right]\right] \\
&= \sum_a \pi(a|s)\sum_{s',r} p(s',r|s,a)[r + \gamma v_\pi(s')]
\end{aligned} \tag{2.1}$$

Gracias a la ecuación de Bellman hemos dado una expresión más interpretable al valor de un estado. Ahora, el valor de un estado es la recompensa que recibimos en ese momento, sumado a una media descontada, (con el factor γ), de todos los posibles estados sucesores, ponderados con la probabilidad de llegar a ellos.

Para resolver la tarea de RL, se busca encontrar una política óptima que maximice este valor del estado, siguiendo la ecuación de Bellman. Hay muchas metodologías para encontrar esta política óptima, pero antes de entrar en ellas, vamos a definir el concepto de optimalidad.

2.4.4. Valores y políticas óptimos

Decimos que una política π es mejor que otra π', si su retorno esperado es mayor o igual que el de π', para todos los estados [18]. Es decir:

$$\pi \geq \pi' \Leftrightarrow v_\pi(s) \geq v_{\pi'}(s), \forall s \in S$$

Decimos, que una política es **óptima**, si es mejor o igual que todas las demás. Puede haber más de una política óptima, pero denotamos a todas ellas con el símbolo π_*.

Todas ellas, siguen funciones de valor óptimas, ya sean de estado:

$$v_*(s) = \max_\pi v_\pi(s) \forall s \in S,$$

ó de estado-acción:

$$q_*(s,a) = \max_\pi q_\pi(s,a) \forall s \in S$$

Además, podemos aplicar la ecuación de Bellman a ambas expresiones, obteniendo la llamada ecuación de Bellman óptima:

$$v_*(s) = \max_{a \in A(s)} \sum_{s',r} p(s',r|s,a)[r + \gamma v_*(s')]$$

$$q_*(s,a) = \mathbb{E}\left[R_{t+1} + \gamma \max_a q_*(S_{t+1},a') \middle| S_t = s, A_t = a\right] = \sum_{s',r} p(s',r|s,a)[r + \gamma \max_{a'} q_*(s',a')]$$

$$q_*(s,a) = \sum_{s',r} p(s',r|s,a)[r + \gamma \max_{a'} q_*(s',a')]$$

Definir estos valores óptimos es necesario ya que, al extraer las funciones de valor óptimas, extraer una política óptima es trivial. En cada paso temporal, habrá una o más acciones que maximicen el valor, cualquier política que priorice esas acciones, será por tanto una política óptima.

En la práctica, calcular la política óptima puede no ser viable en términos de computabilidad. Es por ello, que la mayoría de algoritmos lo que extraen es una aproximación de la política óptima. Algunos de estos métodos, son métodos basados en Programación Dinámica [19], cómo Policy Iteration, ó Value Iteration; métodos de Monte Carlo [20]; ó métodos basados en Aprendizaje de Diferencias Temporales, cómo Q-Learning [21] [22] [23], el método en el que se basa este trabajo.

2.5. Q Learning

Q-Learning es un algoritmo de Aprendizaje por Refuerzo, que busca estimar la función Q a través de la experiencia para encontrar una política óptima. Esta función Q es la función de valor estado-acción que hemos introducido en apartados anteriores. A partir del valor óptimo, q_*:

$$q_*(s,a) = \mathbb{E}\left[R_{t+1} + \gamma \max_{a} q_*(S_{t+1}, a') \middle| S_t = s, A_t = a\right],$$

el algoritmo estima la función Q, a través de la siguiente regla de actualización [23]:

$$Q(s,a) \leftarrow Q(s,a) + \alpha\left[r + \gamma max_{a'} Q(s',a') - Q(s,a)\right],$$

donde α es la tasa de aprendizaje.

Para estimar esta función Q, el algoritmo clásico utiliza una tabla de tamaño SxA, es decir, número de estados por número de acciones. En cada entrada en la tabla, se almacena el valor Q correspondiente a ese estado y esa acción. El agente, según va pasando por diferentes estados y diferentes acciones, actualiza las entradas de la tabla iterativamente, convergiendo al valor Q real cuando $t \to \infty$.

2.5.1. Deep Q Learning

El método clásico de Q Learning genera un problema evidente. Cuando tratamos un problema real, y el espacio de estados y/o el de acciones se convierten en espacios gigantes, se vuelve inviable en términos de memoria mantener una tabla tan grande. Es por ello, que en la práctica, el problema se traslada al campo del Deep Learning, donde en vez de calcular la función Q real, se utiliza un aproximador funcional, para estimarla. Para ello, se utiliza el método de Deep Q Learning (DQN).

Ahora, en vez de estimar los valores Q de nuevos estados, a partir de los anteriores, utilizamos una red neuronal profunda, que aproxima la función Q. La ventaja de este método, es que ya no necesitamos una tabla de tamaño SxA, tan solo necesitamos una red neuronal con S neuronas de entrada, las neuronas ocultas que decidamos utilizar, y A neuronas de salida.

2.5.1.1. Double Deep Q Learning

Con el algoritmo original de Q Learning surgieron varios problemas que llevaron a la comunidad científica a buscar variaciones. La primera que apareció fue la variante del Double Deep Q Learning [24], de *Hado van Hasselt*. Este algoritmo se presentó como una solución al problema de sobreestimación presente en el Q Learning clásico. Consideremos

el valor objetivo de la función Q:

$$Q^*(s,a) = \mathbb{E}_{s'}[r + \gamma \max_{a'} Q^*(s',a')\Big| s,a]$$

En concreto, el problema reside en:

$$\max_{a'} Q^*(s',a')$$

Q Learning es un algoritmo que utiliza estimaciones para calcular otras. Es por ello, que en las fases tempranas de entrenamiento, cuando hay mucho ruido y apenas se han visto ejemplos que hagan que estas estimaciones sean fiables, se está introduciendo una sobreestimación en esa expresión. Esto hace que nuestro sistema introduzca un sesgo hacia los valores máximos en el aprendizaje, lo cuál es un problema muy importante.

Para explicarlo mejor, consideremos un ejemplo: un estado tiene todos los valores Q reales de sus acciones a 0. Sin embargo, los valores estimados están distribuidos por encima de 0. Al tomar el valor máximo de estas estimaciones, estamos actualizando nuestras estimaciones con un valor positivo, lo que hace que estemos sobreestimando la función Q.

La solución propuesta en [24], Double Deep Q Learning, propone la utilización de un doble estimador, donde cada uno es usado para actualizar el otro. En concreto, tenemos un modelo Q, y un modelo objetivo Q'. El algoritmo usa Q' para seleccionar la acción, y Q para evaluar esa acción (estimar los valores Q de la acción). La formulación es la siguiente:

$$Q^*(s_t,a_t) \approx r(s_t,a_t) + \gamma Q(s_{t+1}, \arg\max_{a_{t+1}} Q'(s_t,a_t))$$

El modelo Q se encarga de estimar $\arg\max_{a_{t+1}} Q'(s_t,a_t)$, y el modelo Q' utiliza el valor anterior para estimar $Q(s_{t+1}, \arg\max_{a_{t+1}} Q'(s_t,a_t))$.

Fujimoto et al. profundiza aún mas y propone la variación Clipped Double Q Learning en [25]. Se sigue la propuesta original del Double Deep Q Learning. De nuevo, se calculan dos estimaciones por separado del valor Q real. La distinción se encuentra al calcular Q^*. Para ello, se coge el mínimo de los dos valores Q de los dos estados siguientes (junto a sus respectivas acciones), producidos por los dos estimadores independientes. Cogiendo el mínimo de los valores Q estimados, evitan la sobreestimación. Este método tiene una ventaja importante, mediante esta minimización, se actualizan antes estados con baja varianza en sus estimaciones, lo que lleva a obtener políticas más controladas, y a realizar actualizaciones más seguras.

2.5.1.2. Dueling Deep Q Learning

Con el algoritmo clásico de Q Learning surge otro nuevo problema. Si conocemos que un estado es intrínsecamente malo, ¿por qué deberíamos estimar el valor de las acciones, dado ese estado? En el contexto de los videojuegos, un ejemplo muy claro es un estado en el que sin importar la acción que tomemos, nuestro personaje va a morir.

Para atajar este planteamiento, se propone el algoritmo de Dueling Deep Q Learning [26]. De nuevo, la solución vuelve a ser dividir la tarea a realizar en dos estimadores. Uno de ellos se encarga de estimar el valor de un estado, y el otro se encarga de estimar el valor de una acción, dado ese estado.

La arquitectura del modelo es una red neuronal, donde en la entrada hay una serie de capas ocultas, al igual que en una arquitectura clásica de Deep Q Learning. A partir de ahí, dividimos la red en dos flujos diferentes: uno para cada estimador. Por último, añadimos otra capa que se encarga de agregar los resultados de ambos estimadores. Como la entrada y la salida de la red neuronal, es la misma que la del modelo clásico del Deep Q Learning, podemos entrenar este modelo igual que hacíamos con el original. Es decir, se puede realizar un descenso por gradiente, minimizando una función de error cuadrático, y la formulación será idéntica a la del problema original.

Volviendo a la agregación final, ésta puede realizarse de diferentes maneras. En [26] se propone primero realizar una agregación simple, sumando ambos resultados:

$$Q(s_t, a_t) = V(s_t) + A(s_t, a_t)$$

Sin embargo, los autores acaban identificando a la suma simple como una agregación problemática, ya que sumarlos directamente hace que, a partir del resultado Q, no se pueda recuperar el valor de V y A por separado. En los experimentos, muestran como esta falta de "trazabilidad" resulta en unos resultados muy pobres.

Por tanto, se propone una nueva agregación para la capa final, donde se fuerza que el valor Q sea igual a V, en el caso de que la acción sea aquella con el valor máximo A:

$$Q(s_t, a_t) = V(s_t) + (A(s_t, a_t) - \max_{a' \in |A|} A(s_t, a_t))$$

El artículo profundiza en nuevas agregaciones, pero para el propósito de este trabajo, con las que hemos definido es suficiente para dar contexto al método del Dueling Deep Q Learning.

3

Sistema, diseño y desarrollo

El objetivo de este trabajo es implementar un Sistema de Recomendación basado en técnicas de Aprendizaje por Refuerzo, y observar su eficacia comparándolo con otros algoritmos ya instaurados en el área.

Se ha realizado un considerable trabajo de desarrollo, implementando todos los sistemas desde cero, con una excepción importante, donde se han usado recursos externos: construir el agente de RL utilizando la librería de Keras para RL [1].

3.1. Simulación del ciclo recursivo de recomendación

Independientemente de la técnica que se utilice para generar la recomendación, necesitamos poner a punto un sistema que simule el ciclo recursivo clásico de los sistemas de recomendación: el sistema genera unas recomendaciones que presenta al usuario; éste, según su propio criterio, aporta un feedback sobre esas recomendaciones; por último, el sistema incorpora el feedback del usuario a su conocimiento. Si utilizáramos un sistema de evaluación online, donde contáramos con usuarios reales, este ciclo se implementaría de manera interactiva, ya que el usuario nos daría su feedback real. Sin embargo, para este trabajo no se ha contado con usuarios reales, y se ha realizado una evaluación offline, donde contamos con un conjunto de datos de preferencias de usuario, de los que extraemos los conjuntos de train y test para entrenar y evaluar el recomendador. El feedback del usuario podríamos simularlo con modelos complejos, donde asumimos que el usuario solo puntúa los objetos que le gustan (los relevantes), sólo el que más le gusta (el más relevante), o utilizar modelos más complejos [27] [28].

Sin embargo, como el objetivo del trabajo no es profundizar en como modelar este

feedback, ni en perfeccionar la técnica de evaluación, sino probar técnicas de RL en el campo de la recomendación, vamos a utilizar el modelo de feedback más sencillo de todos: suponemos que el usuario puntúa los objetos que están presentes en el conjunto de datos de test. Es decir, se genera una recomendación, y para extraer el feedback que el usuario daría, se buscan en test los objetos recomendados. Si el objeto está presente, incorporamos a los datos de entrada del recomendador la preferencia descubierta, en forma de tripleta usuario-objeto-puntuación.

Por otro lado, hay varias formas de tratar las puntuaciones que no están presentes en test (missing ratings). Una de las aproximaciones más comunes es tratar los missing ratings como puntuaciones no relevantes, ya que los objetos con missing ratings no son mayoritariamente interesantes. De nuevo, aunque hay aproximaciones más complejas, como este no es el objetivo del trabajo, vamos a trabajar con esta suposición sencilla: si el objeto recomendado no tiene puntuación en test, asumimos que no es relevante. Con estas suposiciones, queda definido el sistema de simulación de recomendación, que implementamos para ser utilizado, independientemente de la estrategia utilizada para generar la propia recomendación.

3.2. Sistema de gestión de ratings

El primer paso para poder implementar un Sistema de Recomendación, es trabajar en el manejo de los datos. Estos sistemas suelen trabajar con matrices dispersas de ratings (Sparse Matrix), ya que los usuarios no puntúan a todos los objetos, solo a una pequeñísima fracción de ellos. Para este trabajo, se ha implementado una matriz dispersa utilizando diccionarios anidados. La clave del primer diccionario, es el ID del usuario, y el valor es otro diccionario, cuya clave a su vez es el ID de un objeto, y el valor es la puntuación que ese usuario ha dado al objeto. Además, también mantenemos un diccionario a la inversa, para poder acceder a los usuarios que han puntuado un objeto. En este caso, se invierten las claves de los dos diccionarios, el primer diccionario tiene como clave el ID del objeto, y el segundo tiene como clave el ID del usuario. Inicializarlos a partir del fichero de ratings es trivial. Tan solo recorremos el fichero, y vamos introduciendo los datos en ambos diccionarios a la vez.

También se implementan diversas funciones para realizar particiones de entrenamiento y test. Nos interesa implementar diversas formas de realizar estas particiones para los experimentos que realizaremos más adelante. En concreto, podemos realizar una por usuarios, donde cada usuario pertenezca por entero a una de las particiones, o por puntuaciones, donde un mismo usuario puede tener puntuaciones tanto en entrenamiento como en test.

3.3. Recomendación basada en RL

Primero, vamos a formalizar la tarea de recomendación, en el paradigma del Aprendizaje por Refuerzo. Para ello, formulamos la tarea como un Proceso de Decisión de Markov (MDP). La interacción natural entre un recomendador y un usuario puede ser modelada a través de la interacción entre el entorno y un agente en RL. Un MDP es una tupla, (S, A, P, R), donde se incluyen el espacio de estados S, el espacio de acciones A, la función de transición P, y la función de recompensa R:

- El espacio de estados, S. En un MDP, cada estado debe contener toda la información relevante obtenida hasta ese momento, ya que recordemos que una de las características principales de los MDPs era que cada estado no dependía de los anteriores. Es por ello, que necesitamos que nuestros estados representen todas las preferencias relevantes descubiertas hasta el momento, de un usuario concreto. Por tanto, una elección con sentido es que el estado sea un vector de preferencias s, de un usuario u. En cada paso de la tarea, donde se le recomienda un objeto a un usuario, y este nos indica si le gusta o no, (relevante o no relevante), descubrimos si ese objeto es relevante, y si lo es, lo añadimos a nuestro estado. Por tanto, el estado es un vector de preferencias, que al comenzar la tarea está vacío, y en el que vamos añadiendo los objetos relevantes que descubrimos en cada paso del recomendador.

- El espacio de acciones, A. Las acciones intuitivas en una tarea de recomendación son los objetos disponibles en cada paso del recomendador. El agente decidirá qué objeto recomendar en cada paso temporal al usuario u. Es por ello, que el espacio de acciones se inicializará a aquellos objetos que no se han recomendado aún a ese usuario, y en cada paso, se eliminará de la lista de posibles acciones el objeto recomendado, pues no tiene sentido recomendarle lo mismo a un usuario, si éste ya nos ha dicho si le gustaba o no.

- La función de transición, $P(s, s', a)$. Define las probabilidades de transición entre dos estados, dada una acción. Por lo tanto:

  $$P(s, s', a) = Pr[S_{t+1} = s' | S_t = s, A_t = a]$$

  Sin embargo, no tiene sentido definir por adelantado las probabilidades de transitar entre varios estados en nuestro problema, ya que a la hora de recomendar, el sistema no va a saber de antemano si el objeto recomendado gustará o no al usuario. Sin embargo, lo que sí sabemos, es que si el objeto recomendado es relevante, se transitará al estado que es igual al estado anterior, pero añadiendo ese objeto relevante a la lista. También sabemos, que si el objeto no es relevante, el estado no cambia. Por último, transitar desde un estado a otro, que no sea generado con esta interacción de recomendar un objeto, y añadirlo si es relevante, es imposible.

- La función de recompensa, $R(s, s', a)$. Define la recompensa esperada inmediata por tomar la acción a desde el estado s, transitando al estado s'. Por tanto:

  $$R(s, s', a') = \mathbb{E}[r_{t+1} | S_t = s, A_t = a, S_{t+1} = s']$$

En nuestra tarea, la recompensa será la puntuación que el usuario dé al objeto que le ha sido recomendado.

3.4. Resolviendo el MDP: Programación Dinámica

Una vez formalizada la tarea de recomendación como un MDP, el siguiente paso lógico es proponer un algoritmo para resolverlo. Recordemos que resolver un MDP, implica encontrar una política óptima, $\pi_*(s)$. Para encontrarla, utilizamos como guía una función de valor, en este caso, también óptima, v_*. De hecho, encontrar una política óptima, conocida una función de valor óptima, es trivial, ya que basta con escoger la acción que maximiza el valor en cada paso del algoritmo. Es por ello que, en la práctica, los algoritmos de resolución de MDPs se centran en encontrar funciones de valor óptimas, ya sean funciones de valor de los estados, $v_*(s)$, o de estado-acción, $q_*(s, a)$.

Una de las principales ramas de algoritmos de resolución de MDPs es la rama de Programación Dinámica, o *Dynamic Programming*. Dentro de ésta, se encuentran los algoritmos de Policy Iteration, y Value Iteration [18]. Dentro del proyecto de investigación de este TFM, se decidió comenzar con el estudio del campo de RL implementando estos algoritmos, ya que son los más simples, y por tanto, son un buen punto de partida.

3.4.1. Policy Iteration

El primer paso para implementar el algoritmo de Policy Iteration, es poder evaluar una política arbitraria. Para ello, dada una política π, necesitamos computar la función de valor del estado, v_π. Este procedimiento recibe el nombre de Policy Evaluation en la literatura [18]. Este proceso es iterativo, se inicializan los valores de todos los estados a 0, para, posteriormente, recorrer todos los estados en cada iteración, actualizando sus valores en cada paso. Esto se repite hasta que la variación entre iteraciones alcanza un mínimo deseable. Este proceso se presenta en el Algoritmo 1.

Una vez definido el algoritmo mediante el cuál podemos evaluar una política, el siguiente paso natural es buscar cómo podemos mejorarla. La mínima variación que podemos hacer en una política es cambiarle sólo uno de sus elementos, es decir, cambiar la acción que tomamos en uno de los estados. Si cambiamos la política, necesitamos saber si este cambio ha supuesto una mejora o un retroceso en la calidad de la misma. Para ello, podemos utilizar el teorema presentado en [18], *policy improvement theorem*.

Dadas dos políticas, π y π', que cumplen la siguiente desigualdad, $\forall s \in S$:

$$q_\pi(s, \pi'(s)) \geq v_\pi(s)$$

Entonces, la política π' debe ser, mejor, o como mínimo, igual, que la política π. Formalizado matemáticamente, esto quiere decir que el retorno esperado debe ser igual o

mayor, $\forall s \in S$:

$$v_{\pi'}(s) \geq v_\pi(s)$$

Este teorema también se cumple para nuestro ejemplo, donde tenemos una política π, y otra idéntica, π', exceptuando uno de los estados, donde hemos modificado la acción a tomar, $\pi(s) = a \neq \pi'(s)$. Como el valor sólo ha cambiado para el estado en el que hemos realizado esa modificación, nos basta con comprobar si $q_{\pi'}(s,a) > v_\pi(s)$. Si esta desigualdad se cumple, entonces podemos afirmar que la política modificada, π', es mejor que π. Este paso recibe el nombre de policy improvement.

El algoritmo de Policy Iteration combina ambas fases, policy evaluation y policy improvement. En cada paso del algoritmo, se evalúa la política actual, se mejora aplicando una fase de policy improvement, y se vuelve a evaluar. En cada paso de policy improvement se garantiza que la nueva política es estrictamente mejor que la anterior, a no ser que la política que teníamos ya fuera óptima, en cuyo caso la nueva puede obtener el mismo rendimiento. En el algoritmo 2 se presenta este proceso.

Algorithm 1: Policy Evaluation

```
   Data: Policy π to be evaluated, variation threshold θ
   Result: Value function V_π(s)
 1 Initialize array V(s) = 0, ∀s ∈ S;
 2 Initialize Δ = θ;
 3 while Δ ≥ θ do
 4 |   Δ ← 0;
 5 |   foreach s ∈ S do
 6 |   |   v ← V(s);
 7 |   |   V(s) ← Σ_a π(a|s) Σ_{s',r} p(s',r|s,a)[r + γV(s')];
 8 |   |   Δ ← máx(Δ, |v − V(s)|);
 9 |   end
10 end
11 return array V
```

3.4.2. Value Iteration

El algoritmo de Policy Iteration garantiza encontrar una política óptima. Sin embargo, tiene una desventaja, y es que en cada paso del algoritmo necesitamos evaluar la política. Esta evaluación ralentiza en exceso el proceso, ya que nos obliga a iterar múltiples veces sobre todos los estados una vez más para evaluar. Por esto, surge la idea de parar el proceso de Policy Evaluation antes de tiempo. Si realizamos, idealmente, pasos infinitos, sabemos que al final la evaluación obtenida converge al valor real de la política. En la práctica, detenemos el algoritmo cuando la variación entre iteraciones alcanza un mínimo aceptable. Sin embargo, si nuestro objetivo es encontrar una política óptima, no necesitamos llegar

Algorithm 2: Policy Iteration

Data: Variation threshold θ

Result: Policy π, Value function $V_\pi(s)$

1 Initialize arbitrarily $V(s) \in \mathbb{R}, \pi(s) \in A(s)$, $\forall s \in S$;
2 $V = policy_evaluation(\pi, \theta)$;
3 $policy_stable \leftarrow false$;
4 **while** $policy_stable \leftarrow false$ **do**
5     $policy_stable \leftarrow true$;
6   **foreach** $s \in S$ **do**
7         $a \leftarrow \pi(s)$;
8         $\pi(s) \leftarrow \arg\max_a \sum_{s',r} p(s', r|s, a)[r + \gamma V(s')]$;
9   **if** $a \neq \pi(s)$ **then**
10             $policy_stable \leftarrow false$
11   **end**
12   **end**
13 **end**
14 **return** *array V, policy* π

a obtener una aproximación tan precisa. En concreto, si realizamos solo una pasada por todos los estados a la hora de evaluar, también está asegurada la convergencia. Este caso concreto, conforma el algoritmo de *Value Iteration*, presentado en el Algoritmo 3. Podemos pensar que Value Iteration intercala un paso de Policy Evaluation y otro de Policy Iteration, pasando por todos los estados en cada barrido. Con esta aproximación, se alcanza la convergencia con más rapidez que en Policy Iteration, y es por ello que en el área de Dynamic Programming se suele utilizar este algoritmo como opción más eficaz.

Hasta ahora hemos presentado algoritmos pertenecientes a la rama de Dynamic Programming. Esta elección se realizó en una fase temprana de la investigación, sin embargo, estos algoritmos no se adecuan al paradigma de la recomendación. Para estos algoritmos, necesitamos saber de antemano las recompensas asociadas a cada par de estado-acción, al igual que las probabilidades de transitar entre estados, (es decir, necesitamos conocer el modelo). Como fue obvio una vez entendido este punto clave, un problema de recomendación no se adecua a estas condiciones. No sabemos la recompensa de la acción hasta que la realizamos, pues no sabemos si le gustará el objeto que le hemos recomendado al usuario hasta que realizamos efectivamente esa recomendación. De aquí también deducimos que no conocemos las probabilidades de transición a priori, pues de si le gusta o no el objeto al usuario, depende si transitamos a un nuevo estado, (recordemos la elección de estados y acciones presentadas en la Sección 3.3). Por estos motivos, el siguiente paso en la investigación fue investigar algoritmos libres de modelo. Entre ellos, uno de los más utilizados, perteneciente a la rama de algoritmos basados en Diferencias Temporales [18], es el algoritmo de Q-Learning.

Algorithm 3: Value Iteration

Data: Variation threshold θ

Result: Policy π

```
1  Initialize arbitrarily V(s) ∈ ℝ, ∀s ∈ S;
2  Initialize Δ = 0;
3  while Δ ≥ θ do
4  |   Δ ← 0;
5  |   foreach s ∈ S do
6  |   |   v ← V(s);
7  |   |   V(s) ← máx_a Σ_{s',r} p(s', r|s, a)[r + γV(s')];
8  |   |   Δ ← máx(Δ, |v − V(s)|);
9  |   end
10 end
11 foreach s ∈ S do
12 |   π(s) = arg máx_a Σ_{s',r} p(s', r|s, a)[r + γV(s')];
13 end
14 return policy π
```

3.5. Resolviendo el MDP: Q-Learning

El algoritmo de Q-Learning, se trata de un algoritmo libre de modelo, donde aprendemos una función de valor estado-acción, Q, que se aproxima a una óptima, q_*. Lo presentamos directamente en el Algoritmo 4. Ahora, la regla de actualización es independiente de la política utilizada, aunque sí necesitamos una para escoger que acción explorar en cada momento. Aquí, introducimos el concepto de las políticas ϵ-greedy. Este tipo de políticas utilizan un parámetro ϵ, para alternar entre exploración y explotación. Cada vez que queramos obtener la acción a ejecutar, dado un estado, se escoge una acción aleatoria con probabilidad ϵ, o una acción que maximice la función de valor Q, con probabilidad $1 - \epsilon$. A través de estas actualizaciones, el algoritmo acaba convergiendo a la función de valor óptima, q_*. Con esta función, el agente tan solo tiene que realizar un $\arg\max$ sobre ella para tener una política óptima.

Sin embargo, este algoritmo nos presenta un nuevo problema. Para actualizar la función Q de esta forma, necesitamos almacenar una entrada para cada par estado-acción, es decir, necesitamos una tabla de tamaño SxA. Además, estima el valor de cada par estado-acción por separado, por lo que no tiene capacidad de generalización para lidiar con estados y acciones que todavía no ha visto [18]. Como el espacio de estados y el de acciones son demasiado grandes en un contexto de recomendación, y por tanto, mantener una tabla con todos sus elementos es inviable en términos computacionales, es fácil concluir que este algoritmo es de nuevo insuficiente. Es por ello, que la investigación derivó hacia la implementación de algoritmos de Deep Q Learning, dónde se utilizan redes neuronales para aproximar la función Q.

Algorithm 4: Q-Learning

```
1  Initialize arbitrarily Q(s,a) ∀s ∈ S, a ∈ A(s), Q(terminal-state, :) = 0;
2  foreach episode do
3  |   Initialize s to initial state;
4  |   foreach step in episode do
5  |   |   Choose a policy π, derivable from function Q (ϵ-greedy);
6  |   |   a = π(s);
7  |   |   Observe r, and s′ (next-state);
8  |   |   Q(s,a) ← Q(s,a) + α[r + γ máx_a Q(s′,a) − Q(s,a)];
9  |   |   s ← s′;
10 |   end
11 end
12 return Learned function Q
```

3.5.1. Deep Q Learning

En la propuesta del Deep Q Learning se utiliza una red neuronal cómo aproximador de la función Q. En la literatura, a la hora de resolver la tarea de recomendación utilizando el paradigma del Reinforcement Learning, la alternativa del Deep Q Learning es una de las más utilizadas [21] [22] [29] [30] [23], debido a que cuenta con las ventajas de Q-Learning, evitando el problema de memoria inherente de este algoritmo. En la Figura 3.1 se muestra el esquema de la interacción entre el agente y el entorno en estos sistemas. El agente utiliza una red neuronal para estimar el valor de la función Q, que llamaremos Q-Network. Esta red neuronal toma como entrada el estado actual, y a la salida obtenemos el valor de todas las acciones posibles. Esta red es entrenada como lo haríamos con cualquier otra, presentándola parejas de entrada-salida en la fase de entrenamiento. Durante esta fase, la red minimiza un error cuadrático, utilizando un optimizador de Deep Learning, de los cuales una alternativa podría ser descenso por gradiente [23]. En concreto, se utiliza un error cuadrático medio entre el valor Q predicho, y el valor objetivo, por lo que se trata de un problema de regresión. Sin embargo, no conocemos ese valor objetivo, ya que estamos tratando con un problema de Reinforcement Learning, por lo que se utiliza la definición de la ecuación de Bellman, (introducida en la Sección 2.4.3.1), para extraer ese valor objetivo t:

$$t = r + \gamma \max_a Q(s', a)$$

Puede parecer que se utiliza para el target el propio valor predicho, pero cómo estamos utilizando la recompensa, al final la red acabará convergiendo. Sin embargo, típicamente en el Deep Q Learning se utiliza una arquitectura de doble red. La Q-Network presentada hasta ahora se encarga tanto de calcular el valor predicho de la función Q cómo el valor objetivo, lo que puede provocar que haya mucha divergencia entre ambos valores. Por ello, se puede utilizar una red especializada en el cálculo de cada valor por separado. La Q-Network ahora se encarga solo de predecir los valores, y utilizamos una "target network"

para estimar el valor objetivo. Esta red es esencialmente igual que la Q-Network, pero tiene unos parámetros fijos. Cada N iteraciones, los pesos de la Q-Network se copian a la target network, con lo que conseguimos un entrenamiento más estable, ya que fijamos la función objetivo durante esas N iteraciones.

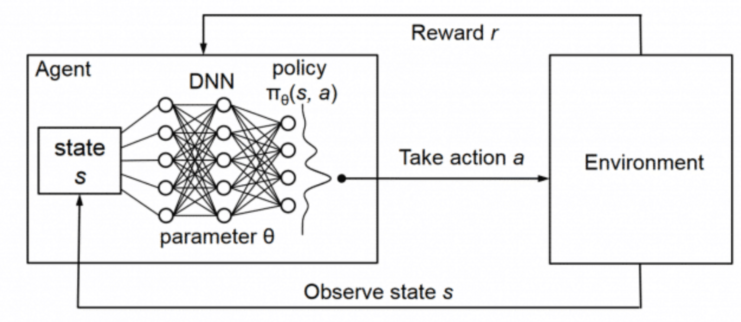

Figura 3.1: Esquemático de un modelo de Deep Q-Learning. Imagen extraída de [31].

3.6. Implementación de Recomendador basado en Deep Q-Learning

Hemos explicado el proceso de investigación que se ha realizado durante el trabajo, y que acabó derivando hacia la implementación de un recomendador basado en Deep Q-Learning. El objetivo ahora es, por tanto, buscar una implementación efectiva, que nos permita contrastar resultados entre diferentes algoritmos de recomendación. Para esta implementación se ha utilizado la librería de Keras-RL [1], que implementa algoritmos de Deep Reinforcement Learning, y los integra con la librería de Python de Deep Learning, Keras. Esta librería nos permite abstraernos de la interacción entre las piezas y de especificar el agente. Tan solo tenemos que construir un entorno con la librería OpenAI Gym [32], una política y un modelo Keras para ser usado como Q-Network.

En la Sección 3.3 ya definimos nuestros estados y acciones. Sin embargo, necesitamos adaptar esas definiciones al nuevo paradigma, ya que ahora los estados necesitan actuar como entrada a nuestra Q-Network. Queremos mantener la definición del estado como una lista de preferencias descubiertas para un usuario u en el instante t, pero necesitamos una representación que nos permita utilizarlo como entrada a la red. En la literatura se proponen diversas soluciones, pero nos acabamos decantando por la propuesta en [23], por ser la más aplicable al contexto del trabajo y ser una solución intuitiva a la vez que elegante. Los autores proponen dos puntos claves:

- Primero, utilizar una aproximación con múltiples MDPs (entornos), uno para cada usuario del sistema. Los estados son dependientes del usuario, ya que la lista de preferencias aprendidas es diferente para cada uno de ellos, por lo que solo hay un usuario asignado a cada entorno. Además, los algoritmos de aprendizaje de Reinforcement Learning se basan en la experiencia, necesitando pasar por una gran

cantidad de pares estado-acción para que el aprendizaje sea efectivo. Esto hace que sea prácticamente imposible encontrar un solo usuario cuya experiencia pueda servirnos para generalizar sobre todo el sistema. Además, utilizando un sólo entorno no se modelarían las relaciones entre usuarios, que aportan muchísima información a la hora de realizar recomendaciones. Es por esto, que el agente va a tratar con múltiples entornos, uno para cada usuario, y extraerá una política a partir de la experiencia de todos ellos. La clave entonces está, en como modelar las relaciones entre ellos, manteniendo las similitudes y diferencias entre los usuarios de cada uno.

- Como consecuencia directa de la aproximación multi-entorno, se propone el uso de estados latentes. Utilizando algoritmos de factorización de matrices, extraemos unas features para crear vectores de características que representen a nuestros usuarios y objetos. Este espacio vectorial es compartido por todos los usuarios y objetos, y modela de manera eficaz las relaciones y las preferencias de los usuarios. Por tanto, nuestra nueva representación para los estados es el vector de features del usuario u.

Estas dos ideas clave definen por completo nuestro recomendador basado en Deep Q-Learning. Aunque el sistema se ha propuesto en [23], cabe destacar que se ha implementado por completo, utilizando las librerías ya mencionadas.

3.6.1. Sistema de Estados Latentes

La elección de los vectores de características de cada usuario como estados conlleva que necesitemos un buen sistema, basado en factorización de matrices, para extraer estas features que nos permitan modelar a cada usuario. Además, debemos mantener el espíritu iterativo propio de la recomendación y del Reinforcement Learning. La idea es que, con un conjunto de entrenamiento, se preentrene un modelo de factorización de matrices para extraer los vectores de características de los objetos. Después, durante el proceso iterativo natural de la recomendación, iremos obteniendo nueva información sobre los gustos de los usuarios, y por tanto, sus vectores de características deberán actualizarse de manera online. Para ello, utilizaremos los nuevos ratings descubiertos, además de los vectores de características de los objetos, obtenidos previamente.

Para ello, el primer paso es preentrenar un modelo de factorización de matrices, utilizando la matriz de puntuaciones. El objetivo de este pre-entrenamiento es, cómo ya hemos comentado, obtener los vectores de características de los objetos. Por simplicidad, estos vectores no van a ser actualizados de manera online. Esto implica que el sistema conoce desde el principio todos los objetos que van a ser utilizados, es decir, no contemplamos la opción de añadir nuevos objetos una vez comenzado el aprendizaje. Como optimizador utilizamos descenso por gradiente, minimizando el error cuadrático. Las reglas de actualización [23], para los vectores de features de usuarios y objetos, son las siguientes:

$$U_u \leftarrow U_u - 2\alpha\left[(U_u^T V_i - R_{ui})V_i + \lambda U_u\right]$$

$$V_i \leftarrow V_i - 2\alpha\left[(U_u^T V_i - R_{ui})U_u + \lambda V_i\right],$$

dónde λ es el factor de regularización, y α es la tasa de aprendizaje. Utilizando estas reglas de actualización, recorremos la matriz de ratings R, actualizando ambos vectores para cada usuario y objeto. La configuración del sistema se compone del número de features, K, el numero de iteraciones, I, y el valor de α y λ. En nuestra implementación, $K = 16$, $I = 20$, $\alpha = 0{,}1$ y $\lambda = 0{,}01$. Así, fijamos los vectores de features de los objetos, V, y los utilizamos para los siguientes pasos.

Ahora, tan solo tenemos que definir cómo obtenemos los estados latentes, y como los actualizamos durante el proceso iterativo del Reinforcement Learning. Para empezar, en la primera iteración, inicializaremos los vectores de features de los usuarios a ceros. La motivación de esta inicialización, es que nuestro estado era la lista de preferencias descubiertas para el usuario u, y en el instante $t = 0$, esta lista está vacía, por lo que una inicialización a un vector nulo parece lo más conveniente. En sucesivas iteraciones, mientras el agente interactúa con el entorno, cuando descubramos un nuevo rating aplicaremos la regla de actualización de U_u para actualizar el vector de features del usuario u, que compone el estado latente, s_t. Esto se realiza durante todo el proceso de entrenamiento del sistema.

Cómo indican en [23], estos estados latentes mantienen la propiedad de Markov intacta, ya que cada actualización en el instante t, sólo depende del instante $t - 1$. Es decir, el último estado s_t es independiente de los anteriores. Por tanto, podemos seguir aplicando todos los algoritmos y técnicas estándar de Reinforcement Learning vistas hasta ahora.

Para implementar esta parte, no se ha utilizado código plenamente original. Se partió de un factorizador de matrices basado en Descenso por Gradiente ya implementado[1], y se modificó para incluir las actualizaciones online requeridas para nuestros estados latentes.

3.6.2. Modelo de Q-Network

La siguiente pieza clave es el modelo que escogemos para implementar nuestra Q-Network. En Keras-RL, basta con inicializar un modelo de Keras, y pasárselo al agente, para que este lo utilice como Q-Network. En nuestro caso, utilizamos una red con 2 capas ocultas, de 64 neuronas cada una. Utilizamos el optimizador Adam [33], por ser una versión basada en el descenso por gradiente, que añade tasas de aprendizaje adaptativas. El número de entradas es el número de características que usamos, en nuestro caso, 16. Por último, el número de salidas es el número de acciones, es decir, cuántos objetos conoce nuestro sistema. Para MovieLens-100k [34], el número de objetos disponibles en todo el sistema es 1700. En la Figura 3.2 se incluye el esquemático generado con la librería Keras del modelo utilizado. Cabe destacar que se incluye una capa Flatten, entre la entrada y la

[1] Presentado y explicado en `https://www.ethanrosenthal.com/2016/01/09/explicit-matrix-factorization-sgd-als/`, y público en GitHub: `https://gist.github.com/EthanRosenthal/a293bfe8bbe40d5d0995`

primera capa oculta. Se necesita incluir esta capa por la forma en la que está implementado el agente en la librería Keras-RL.

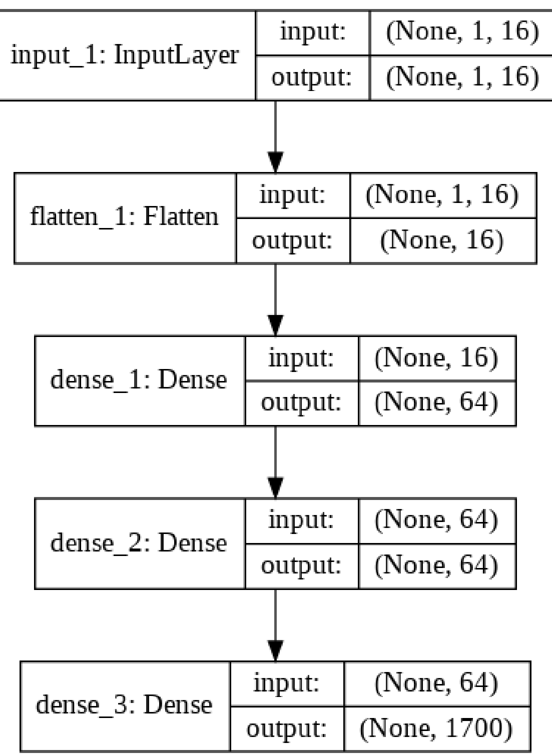

Figura 3.2: Modelo Keras de la Q-Network descrita.

3.6.3. Política utilizada

Necesitamos especificar qué política vamos a utilizar, utilizando los valores Q para decidir qué acción tomar. Una de las políticas más extendidas es la ϵ-greedy, debido a su facilidad de uso y a qué nos permite equilibrar entre exploración y explotación. El parámetro ϵ es el que se encarga de esto. Este tipo de políticas, escogen una acción aleatoria con probabilidad ϵ, ó la acción con mayor valor Q con probabilidad $1-\epsilon$. Aunque esta política sería suficiente, pues es óptima si se utiliza una función de valor óptima, como es la función Q, vamos a incluir ciertas variaciones. La primera, es añadir un ϵ dinámico. Durante el entrenamiento, queremos que al principio sea muy grande, para que se explore más, y según vaya avanzando el proceso, ϵ decrezca para priorizar más la explotación. En la Figura 3.3 se puede observar las áreas de exploración y explotación en función del valor de ϵ. La siguiente, es añadir que cada acción sólo puede ser escogida una sola vez. Esto tiene mucho sentido en el contexto de la recomendación, ya que, para un usuario, no queremos recomendarle el mismo objeto varias veces. Si ya nos indicó si le gustaba, no tiene sentido volver a recomendarlo. Cómo esto tiene sentido sólo para un mismo usuario, y no entre usuarios diferentes, también incluimos la posibilidad de hacer un reset en las acciones posibles. Queremos que cuando cambiemos de usuario, se vuelvan a restaurar las acciones posibles, escogiéndolas con cuidado. Para acelerar el entrenamiento, y ya

que vamos a ignorar los missing ratings, restauramos las acciones posibles al conjunto de objetos que el usuario ha puntuado en el conjunto de datos de entrenamiento.

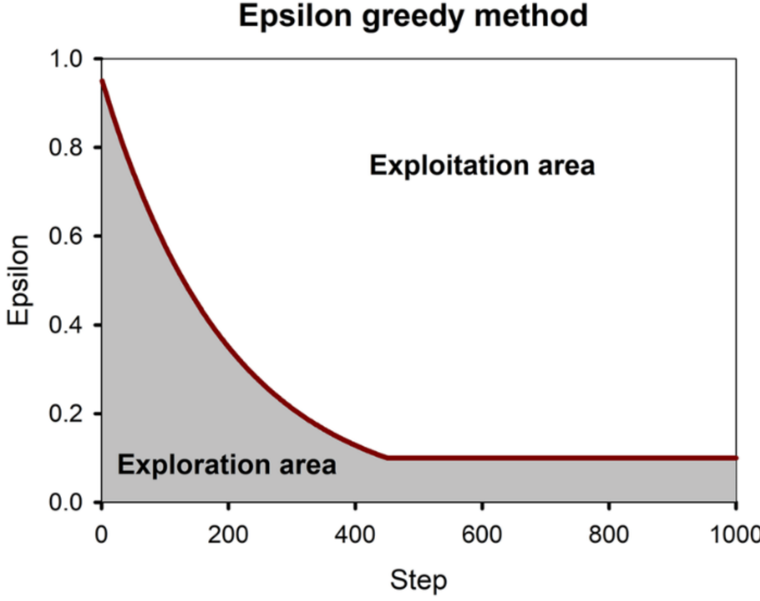

Figura 3.3: Equilibrio entre explotación y exploración, en función del valor de ϵ. Imagen extraída del portal ResearchGate: `https://www.researchgate.net/figure/Epsilon-greedy-method-At-each-step-a-random-number-is-generated-by-the-model-If-th fig2_334741451`

3.6.4. Entorno de estados latentes

El último elemento que necesitamos implementar, es el entorno. Para ello, Keras-RL utiliza la librería OpenAIGym, que proporciona clases y métodos para implementar de manera sencilla y genérica diferentes entornos. Aunque la librería incluye implementaciones propias, nosotros necesitamos especificar el nuestro propio. Lo nombramos como entorno de estados latentes, ya que utiliza el sistema de estados latentes para decidir cómo transitar. Los entornos Gym tienen que extender de la clase Env, y necesitan implementar 4 métodos: uno de inicialización, un step, un reset, y un render.

El step recibe una acción, y devuelve una tupla con el estado siguiente, la recompensa asociada a esa transición, una bandera indicando si ha finalizado el episodio, y un diccionario con datos para debug. El método reset es llamado tras finalizar cada episodio, y el render es utilizado para depurar. En nuestro caso, estábamos utilizando una aproximación multi-entorno, donde cada entorno se asociaba con cada usuario. Además, en cada episodio se le realizan recomendaciones a un solo usuario, por lo que en el reset hay que reiniciar el estado actual a todo ceros, (el estado inicial).

En el método step, buscamos el rating asociado al usuario actual y objeto correspondiente a la acción en la matriz de ratings. Si está presente, actualizamos el estado latente cómo se indicó en la Sección 3.6.1. Si no estaba presente, el estado latente

no se altera, pues no hemos descubierto nuevas preferencias del usuario. La recompensa es el rating descubierto, siendo 0 si el usuario no había puntuado ese objeto en la matriz de ratings. La condición de parada para el episodio es un límite de pasos. Cuando se le han recomendado N objetos, se activa la bandera de final de episodio.

3.6.5. Tipos de Deep Q-Learning

En las Secciones 2.5.1.1 y 2.5.1.2 ya se introdujeron 2 tipos especiales de Deep Q-Learning: Double Deep Q-Learning, y Dueling Deep Q-Learning. En Keras-RL, se incluye la posibilidad de especificar si nuestro agente debe utilizar alguna de estas dos opciones, o usar una Q-Network simple. En concreto, tenemos 3 formas de aplicar el Deep Q-Learning:

- Deep Q-Learning (DQN). Tan solo se usa una red, llamada Q-Network, para estimar los valores de la función Q. Estas mismas estimaciones son usadas para actualizar la red, lo que puede llevar a problemas de overfitting.
- Double Deep Q-Learning (DDQN). Se desglosa la tarea de estimar la función Q, y calcular el valor objetivo (target) en dos estimadores independientes. Cada uno de ellos se actualiza teniendo en cuenta el otro. El algoritmo de Double Deep Q-Learning fue propuesto en [24].
- Dueling Double Q-Network (D3QN). Podemos combinar las técnicas de DDQN y Dueling DQN, para extraer el mayor beneficio. Recordemos que en el Dueling DQN se dividía la tarea de calcular el valor Q en calcular dos valores, el valor del estado, $V(s)$, y la ventaja de tomar una acción en ese estado, $A(a, s)$, para después agregarlos de alguna forma. En [26] se propone el algoritmo y las diferentes opciones para realizar esta agregación.

Para el caso de Dueling DQN, Keras-RL implementa tres posibles agregaciones, presentadas en [26]:

- Agregación simple. La primera propuesta del artículo, simplemente suma ambos términos.
$$Q(s, a) = V(s) + A(s, a)$$
- Agregación por máximo. Con la agregación anterior, no se puede trazar los valores de V y A a partir de Q. Para resolver este problema de trazabilidad, proponen forzar que A se anule en la acción escogida.
$$Q(s, a) = V(s) + (A(s, a) - \max_{a' \in \mathbb{A}} A(s, a'))$$
- Agregación por media. Modificación de la anterior, cambiando el operador del máximo por una media sobre las acciones. Cabe destacar que en el artículo se

presenta esta opción como la más recomendable.

$$Q(s,a) = V(s) + (A(s,a) - \frac{1}{|\mathbb{A}|}\sum_{a'} A(s,a'))$$

En nuestro caso, vamos a utilizar los tres tipos de Deep Q-Learning (DQN, DDQN, y D3QN), utilizando la agregación por media en el último de ellos.

3.6.6. Entrenamiento del recomendador

Juntando todas las piezas que hemos definido por separado, entrenamos nuestro recomendador. En la Figura 3.4 se puede visualizar la arquitectura general, en lo referente a la interacción entre el agente y el entorno. Como ya hemos comentado, la idea de utilizar una aproximación multi-MDP, utilizando vectores de características para representar estados y acciones, está extraída de [23]. Es por ello que la figura que muestran en el artículo nos sirve también para visualizar mejor como funciona nuestro sistema.

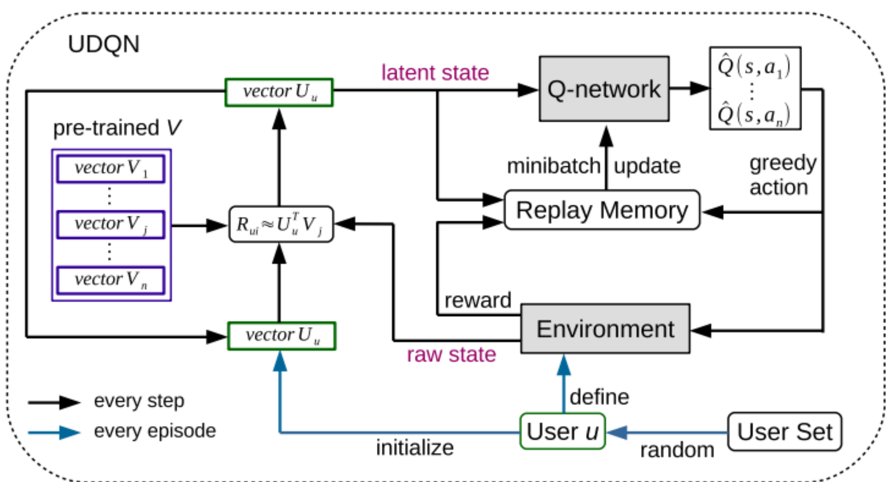

Figura 3.4: Diagrama general del recomendador basado en Reinforcement Learning. Imagen extraída de [23].

Utilizando el Sistema de gestión de ratings, cargamos los ratings de entrenamiento. Inicializamos política y modelo Keras, y se los pasamos al agente Keras-RL. Inicializamos el sistema de estados latentes, y lo precomputamos previamente para obtener los vectores de los objetos. Recorremos los usuarios presentes en el conjunto de entrenamiento, de manera aleatoria, y para cada uno de ellos:

- Reiniciamos las acciones posibles de la política a los objetos puntuados por el usuario actual en el conjunto de entrenamiento.

- Reiniciamos el entorno, indicándole el usuario actual y cuántos objetos queremos recomendar (el límite de pasos por cada episodio).
- Entrenamos la Q-Network, pasándole el nuevo entorno. Se va a entrenar durante tantas iteraciones como objetos queremos recomendar al usuario (el límite del paso anterior).
- Guardamos la recompensa acumulada del episodio, y pasamos al siguiente usuario.

Esta recompensa acumulada es la métrica que vamos a visualizar en los experimentos. Si la matriz de ratings contiene puntuaciones discretas, entre 0 y 5, por ejemplo, no podemos saber cuantos objetos relevantes se han descubierto en cada paso, pues Keras-RL solo devuelve la recompensa acumulada, y no un histórico de cada paso realizado. Por tanto, no podemos saber si una recompensa de 20, por ejemplo, se ha obtenido con 5 pasos donde los ratings descubiertos han sido un cero, y cuatro cincos, donde habría 4 objetos relevantes, o han sido cinco puntuaciones de cuatro, donde habría 5 objetos relevantes (si el umbral de relevancia es la puntuación 4). Sin embargo, podemos transformar nuestros ratings a binarios (rating 0 si es no relevante, rating 1 si es relevante), y entonces sí podríamos extraer esta información, ya que sabemos que cada rating solo puede sumar un 1, y por tanto, la propia recompensa acumulada indica el número de objetos relevantes descubiertos. Ambas opciones tienen sus ventajas, y en los experimentos se indicará cual de ellas se utiliza en cada momento.

4

Resultados

En esta Sección se describirán en detalle los experimentos realizados para probar el funcionamiento de las técnicas de Reinforcement Learning descritas en la Sección anterior. Estos experimentos tienen el objetivo de evaluar el sistema, enfrentando a varias técnicas de Deep Q-Learning primero, para poder escoger la más adecuada y enfrentarla a otros algoritmos de recomendación.

Primero, vamos a detallar la estructura de los conjuntos de datos utilizados, (por ejemplo, la distribución de los usuarios en entrenamiento y en test), para poder entender mejor los resultados que obtendremos más adelante. Después, vamos a definir una tarea experimental clara y concisa. Tras esto, especificaremos la configuración de los parámetros utilizados para ejecutar los algoritmos. Por último, mostraremos los resultados obtenidos tras poner en marcha la tarea experimental definida.

4.1. Detalles de los conjuntos de datos

Aunque el objetivo era realizar experimentos con varios conjuntos de datos, y diferentes configuraciones para poder tener una configuración experimental rica y compleja, la situación vivida por el COVID-19 durante el desarrollo de este TFM ha impedido que se puedan realizar experimentos tan variados. Debido al cierre de la universidad, a la imposibilidad de acceder al equipo de la Escuela Politécnica Superior, y a la situación personal vivida por el estudiante, estos experimentos se han visto muy reducidos. Con el equipo disponible desde casa, las ejecuciones de los experimentos se eternizaban demasiado, es por ello que se ha tenido que reducir la combinatoria a un solo conjunto de datos: MovieLens [34], en su versión 100k (ml-100k), y a una sola partición entrenamiento-test, del 20 % para entrenamiento, y 80 % para test, ya que el entrenamiento es la fase

más lenta, y por tanto, esta opción es la que más agiliza el tiempo de ejecución. El resto de configuraciones y experimentos que se prentendían realizar, quedan pendientes como trabajo futuro, cuando la situación se normalice y se puedan realizar con tiempos de ejecución prudenciales.

Por tanto, tan solo tenemos que analizar un conjunto de datos: MovieLens, en su versión de 100.000 puntuaciones, al que nos referiremos por el nombre ml-100k. Este dataset viene ya particionado por sus creadores, utilizando una partición del 80 % de los datos para entrenamiento y el 20 % para el test. Nosotros queremos utilizar la partición inversa, así que simplemente utilizamos los datos de test como entrenamiento, y viceversa. Este conjunto de datos cuenta con 100.000 puntuaciones, que van del 1 al 5, de 943 usuarios y 1682 películas (objetos), donde cada usuario ha puntuado como mínimo 20 películas. Otra característica importante a visualizar es la cantidad de puntuaciones que tiene cada objeto tanto en la partición de entrenamiento como en test. En la Figura 4.1 podemos observar como tanto los datos de entrenamiento como los de test están claramente sesgados por popularidad. Este sesgo pretende representar el que tendría un conjunto de datos de un sistema real, ya que normalmente hay unos pocos objetos muy populares con muchas más puntuaciones que la gran mayoría del resto de objetos (distribución power-law).

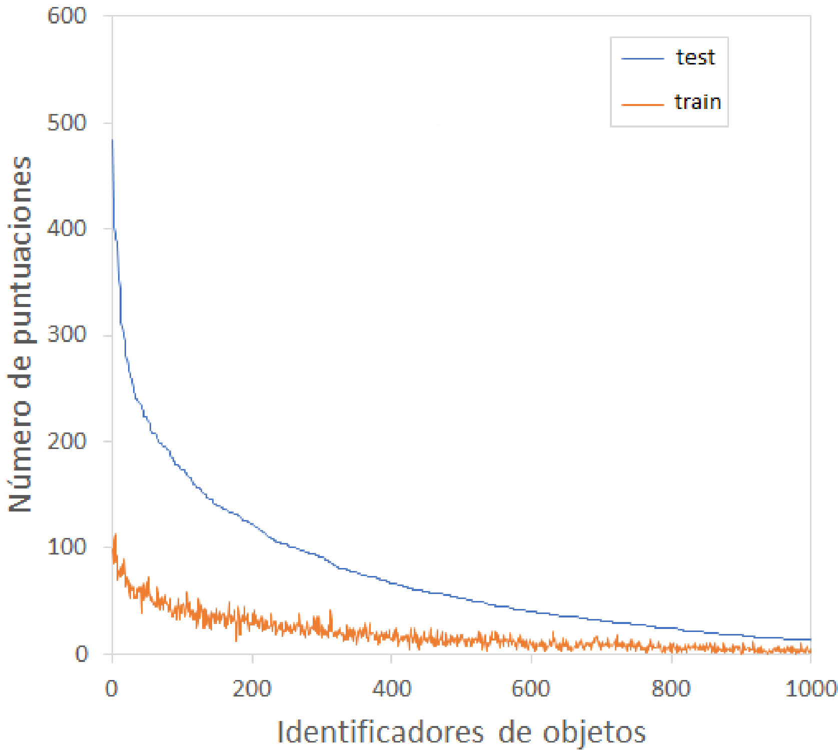

Figura 4.1: Distribución de los objetos en ml-100k.

Cabe también destacar que en el entrenamiento solo aparecen el 48 % de los usuarios, mientras que en test aparecen el 100 %. Es decir, hay 491 usuarios que no tienen puntuaciones en entrenamiento. Esto es muy importante a la hora de entender los resultados que mostraremos en los experimentos.

4.2. Definición de la tarea

Para poder realizar un estudio de las diferentes estrategias de Reinforcement Learning explicadas en la anterior sección, vamos a definir una tarea básica que aplicaremos en los experimentos. Esta tarea consiste en simular de manera simplificada, mediante evaluación offline, el proceso de recomendación continua que se produciría en un sistema real. Esto es, el sistema genera recomendaciones por lotes a cada usuario, estos usuarios devuelven información al sistema mediante el feedback que generan al puntuar esas recomendaciones, y el sistema incorpora esta información a la ya conocida para volver a generar recomendaciones. Vamos a aplicar esta tarea a dos experimentos diferenciados. El primero, tiene como objetivo contrastar resultados entre diferentes técnicas de Deep Q-Learning; mientras que en el segundo, compararemos los resultados de la mejor técnica de Deep Q-Learning, contra diferentes baselines de recomendación.

El carácter temporal de este proceso hace que se planteen diferentes configuraciones temporales para simularlo: el sistema podría encontrarse en un estado de arranque en frío, o podría tener ya información suficiente sobre la gran mayoría de sus usuarios y por tanto disponer de una información poco dispersa. La intención era representar estas dos posibilidades con dos particiones de los conjuntos de datos: una partición 20-80 de entrenamiento-test para el arranque en frío, o una 80-20 para la alternativa avanzada. Sin embargo, como ya se ha comentado, los experimentos se limitan a la partición 20-80. A su vez, dentro de estas particiones, también hay diferentes fases: temprana, intermedia y avanzada, que comprenden respectivamente, al 25 %, 75 % y 100 % de los episodios. Utilizamos esta nomenclatura ya que es importante distinguir entre diferentes momentos del ciclo recursivo, ya que en algunas ocasiones podría interesarnos priorizar más la fase temprana, donde necesitamos mejor rendimiento para no perder a los usuarios, en otras podríamos sacrificar un poco la fase temprana a cambio de mejorías notables en las fases intermedias o avanzadas, etc.

Es importante también mencionar que consideramos que el usuario puntúa de vuelta todo lo que se le recomienda. Se podrían implementar diferentes estrategias de feedback, para simular si el usuario puntúa todos los objetos que le gustan (aquellos que son relevantes), solo el más relevante, etc. Sin embargo, por simplicidad, asumimos que el usuario da feedback para todas las recomendaciones que se le presentan. Esta simulación de feedback se realiza utilizando el conjunto de datos. Cada vez que se le realiza una recomendación a un usuario, se busca si existía esa puntuación en el dataset. Si no existe, asumimos que el usuario lo puntúa con un 0; si existe, lo puntúa con el valor presente en los datos.

Debido a la naturaleza del Deep Q-Learning, necesitamos saber desde el principio cuántas acciones hay disponibles. Es por ello, que necesitamos saber de antemano el número de objetos disponibles, y no puede alterarse. Por tanto, no se contempla la posibilidad de que, una vez arrancado el sistema, se añadan nuevos objetos. Por otro lado, ya hemos comentado que hay usuarios que no cuentan con puntuaciones en entrenamiento, pero si en test. Por tanto, estamos simulando el proceso de llegada de nuevos usuarios.

Por último, es importante aclarar una limitación importante de los experimentos

offline. Con ellos, no podemos obtener información sobre puntuaciones que no hay en el conjunto de datos. Asumimos que esta falta de datos es equivalente a un dato no relevante. Una justificación para asumir esta equivalencia como razonable es que la probabilidad a priori de que un dato sea relevante (dado un usuario y un objeto al azar) es muy baja. Esta limitación es recurrente en todos los experimentos offline realizados en el campo de la recomendación.

4.3. Configuración y parámetros

Debido a la complejidad del sistema, hay multitud de hiperparámetros que pueden modificar el rendimiento del mismo. Todos los parámetros se escogen teniendo en cuenta las configuraciones que realizan los autores en su propuesta del sistema en [23], aunque se han modificado algunos, con la intención de conseguir tiempos de ejecución más razonables. Para el sistema de estados latentes, se utiliza un algoritmo de Factorización de Matrices, utilizando descenso por gradiente. Aquí, se utilizan **vectores de 16 características**, con una **tasa de aprendizaje de** 0,1. En la fase de precomputado, donde se extraen los vectores de features de los objetos, se aplican **20 iteraciones**. Una vez arrancado el sistema, cuando actualizamos los vectores de los usuarios de manera online, utilizando las puntuaciones que se van descubriendo, utilizamos también **20 iteraciones** en cada paso.

Para la arquitectura de la Q-Network, utilizamos una entrada de **16 neuronas**, ya que tiene que coincidir con el número de características (el vector de características es nuestro estado). Utilizamos además **2 capas ocultas**, de **64 neuronas cada una**. Por último, el número de neuronas de salida debe coincidir con el número de objetos diferentes que hay en el dataset. Para Ml-100k, son **1700 neuronas**. Utilizamos un **optimizador Adam**, activaciones **sigmoidales** en las capas ocultas, y **lineal** en la capa de salida. La razón de esto, es que necesitamos que la activación de las capas ocultas sea diferenciable para propagar fácilmente el error. Por otro lado, en la salida es lineal ya que estamos resolviendo un problema de regresión, donde queremos aproximar la función Q.

Para la política, utilizamos un ϵ dinámico en entrenamiento, que varía desde 1 **hasta** 0,1 **en** 10000 **pasos**. En test, utilizamos un ϵ estático, cuyo valor es 0,05.

Si ejecutamos el D3QN, utilizamos la agregación por media vista en la Sección 3.6.5. A cada usuario le recomendamos 40 objetos, ya que en [23] proponen recomendar a cada usuario la mitad del número de puntuaciones que tiene cada uno en los datos de entrenamiento. Sin embargo, por simplicidad a la hora de comparar con otros algoritmos de recomendación, asignamos un valor global. En los experimentos se observó que la mayoria de usuarios recibían un número de recomendaciones cercanas a 40, por lo que se fijó a este valor directamente. Estos usuarios se recorren de manera aleatoria tanto en entrenamiento como en test.

Como métricas, vamos a utilizar la recompensa acumulada en entrenamiento, y la relevancia descubierta acumulada, ó recall en test. Utilizamos la recompensa acumulada, ya que la recompensa por episodio fluctúa demasiado, haciendo que las gráficas sean

difíciles de interpretar. En test, si dividimos esta recompensa acumulada por el número de objetos relevantes presentes en el dataset, podemos obtener el Recall que hemos acumulado. Esto es interesante para poder visualizar cómo de rápido descubre nuestro agente la relevancia presente en el conjunto de datos.

4.3.1. Barrido de gamma

Para el agente, necesitamos escoger un factor de descuento γ. Recordemos que este parámetro regula la importancia de las recompensas futuras, siendo estas más importantes cuanto más se aproxime a 1. Aunque podríamos escoger un valor en base a la intuición, es más interesante realizar un barrido sobre diferentes valores y observar cuál obtiene mejores resultados. Para no eternizar la ejecución, utilizamos una versión reducida del dataset de MovieLens-100k, utilizando su partición del 20 % para entrenamiento y 80 % para test. Nos quedamos con el 10 % de usuarios tanto en entrenamiento como en test, escogidos de manera aleatoria. Así, construimos un dataset que vamos a nombrar MovieLens-10k. Sobre este dataset, vamos a entrenar el agente con diferentes valores de gamma, y observar qué resultados obtenemos, utilizando las métricas mencionadas. El barrido se realiza sobre $\gamma = [0{,}1, 0{,}2, 0{,}3, 0{,}4, 0{,}6, 0{,}8, 0{,}9]$.

En las Figuras 4.2 y 4.3 se pueden visualizar los resultados tanto en entrenamiento como en test para cada valor de γ. Además, en la Tabla 4.1 se presentan las recompensas medias en test. A la vista de estos resultados, la elección de γ no es trivial. Algunos valores obtienen mejores resultados en los primeros episodios, pero son superadas por otras opciones mejores cuanto más avanzamos en el tiempo. Podría parecer que si tenemos que esperar muchos episodios para obtener buenos resultados, podríamos perder a los usuarios por el camino, y por tanto, deberíamos priorizar los primeros episodios. Sin embargo, recordemos que cada episodio es un solo usuario, al que se le recomiendan 40 objetos, por lo que el compromiso no es tan grande. En entrenamiento, $\gamma = 0{,}1$ obtiene los mejores resultados durante la mayoría de episodios. Sin embargo, en test, $\gamma = 0{,}9$ y $\gamma = 0{,}4$ están muy igualados. Debido a que $\gamma = 0{,}4$ obtiene la mejor recompensa media en test, y que si utilizáramos un $\gamma = 0{,}9$ las recompensas inmediatas no tendrían apenas importancia, decidimos que el mejor candidato es $\gamma = 0{,}4$.

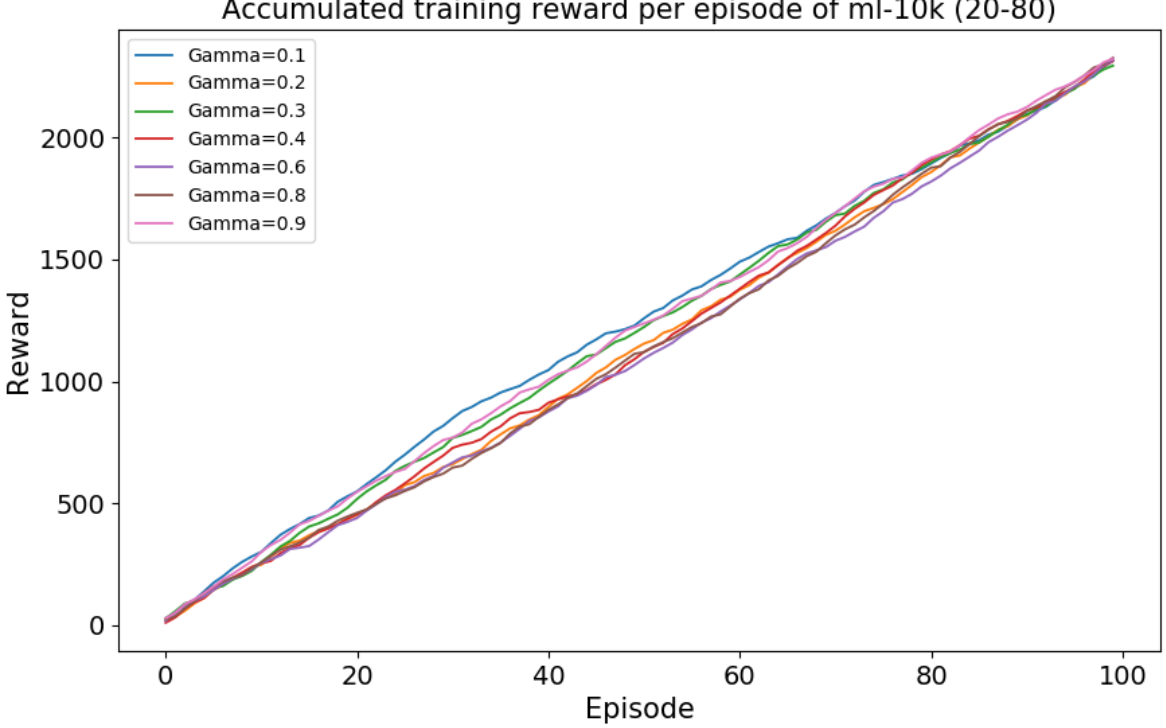

Figura 4.2: Recompensa acumulada por episodio en la fase de entrenamiento para el barrido de γ.

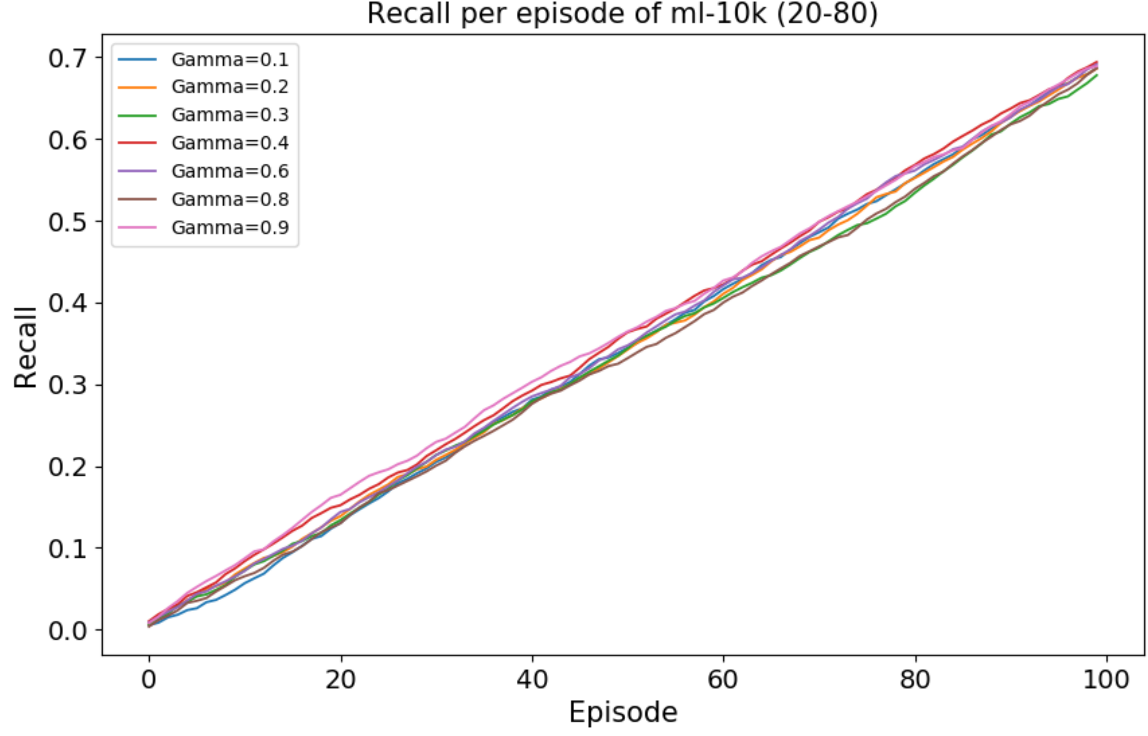

Figura 4.3: Recall por episodio en la fase de test para el barrido de γ.

Gamma	Mean Reward (Test)
0.1	24.22
0.2	24.19
0.3	23.9
0.4	24.47
0.6	24.37
0.8	24.2
0.9	24.29

Tabla 4.1: Mean Reward en test para cada valor de γ.

4.4. Comparativa de técnicas de Deep Q-Learning

Una vez definida la configuración de los parámetros, vamos a comparar el rendimiento de las tres alternativas de Deep Q-Learning. En concreto, vamos a comparar las técnicas planteadas en el Apartado 3.6.5. Como se trata de un problema de Reinforcement Learning, vamos a evaluar la recompensa acumulada del agente a través de los episodios realizados durante el proceso interactivo. Para poder visualizar claramente las diferencias entre las diferentes técnicas, vamos a restar a cada curva la recompensa obtenida con una

recomendación aleatoria. Se muestra la recompensa acumulada en la fase de entrenamiento en la Figura 4.4, y la correspondiente a la fase de test en la Figura 4.5.

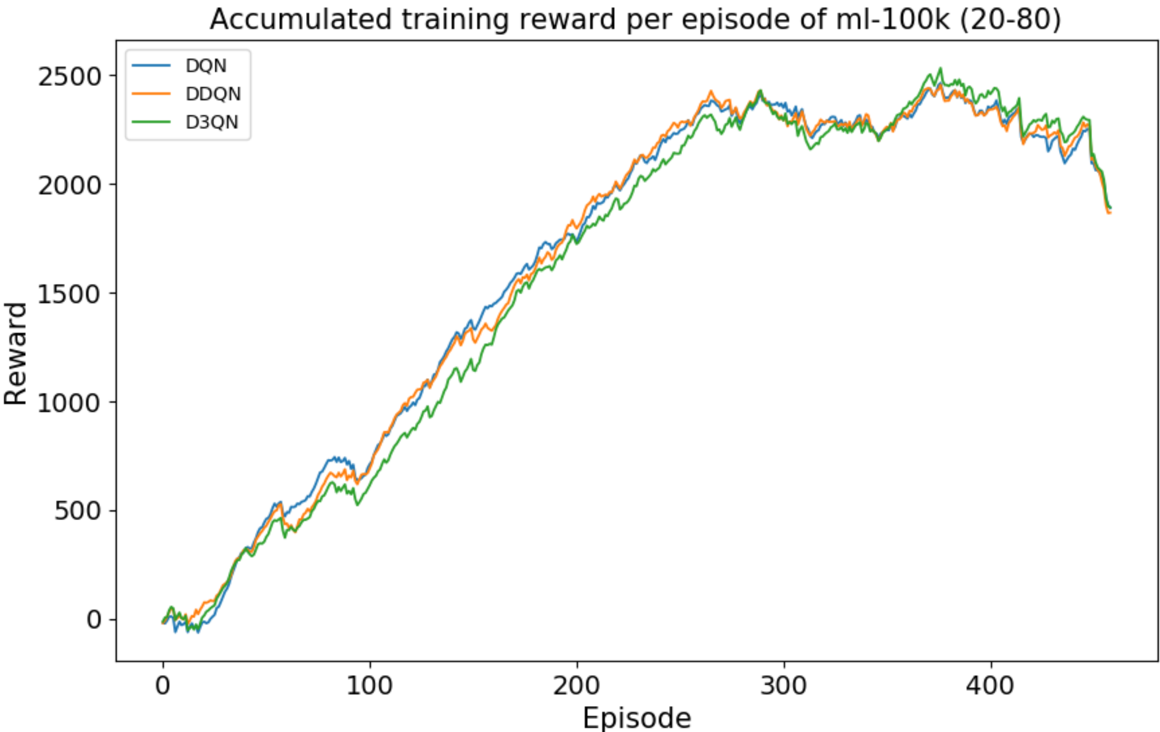

Figura 4.4: Recompensa acumulada por episodio en la fase de entrenamiento para la comparativa de técnicas de Deep Q-Learning.

Lo primero que sorprende es la gran similitud que existe en los resultados de las tres alternativas. Una posible explicación para esto es el uso de los estados latentes. Cómo modelamos las preferencias de los usuarios, y las relaciones entre ellos con factorización de matrices, ya tenemos una buena codificación que engloba todas esas relaciones, y aunque usemos modelos más complejos de Deep Q-Learning, el margen de mejora no es tan grande.

En la fase temprana, D3QN es el que peores resultados obtiene, viéndose superado por DQN y DDQN, aunque alcanza mejores resultados en la fase avanzada. Por otro lado, DQN cuenta con la ventaja de ser la técnica más sencilla, aunque a partir del episodio 200 se ve superado por DDQN y D3QN. En test, DDQN es el que mejores resultados obtiene en la fase temprana, aunque en la intermedia el mejor es D3QN, que da paso a DQN como mejor alternativa en la fase tardía. A la vista de los resultados, no es fácil dilucidar cual es la mejor opción. En la Tabla 4.2 se muestran las recompensas medias para cada algoritmo en entrenamiento y en test. Cómo los resultados no son concluyentes, se escoge el algoritmo DDQN para ser comparado con baselines de recomendación, ya que es una opción intermedia tanto en complejidad como en rendimiento.

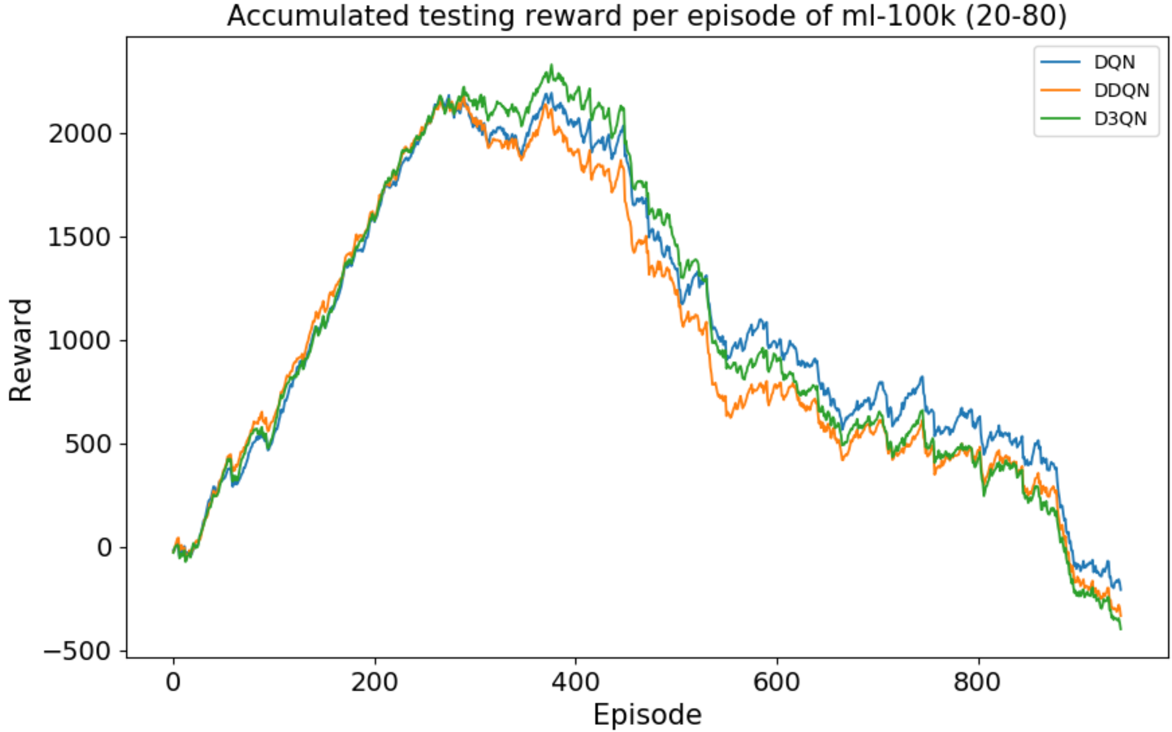

Figura 4.5: Recall por episodio en la fase de test para la comparativa de técnicas de Deep Q-Learning.

Algoritmo	Mean Reward (Train)	Mean Reward (Test)
DQN	23.675	23.13
DDQN	23.627	22.99
D3QN	23.68	22.93

Tabla 4.2: Mean Reward en test para cada valor de γ.

4.5. Comparativa de algoritmos de recomendación

Una vez escogido DDQN como algoritmo candidato de Reinforcement Learning, vamos a comparar con diferentes estrategias de recomendación. La intención es poder comparar con algoritmos de cada área principal, es decir, vecinos próximos (kNN basado en usuario, **ub**), factorización de matrices (**pzt** [35]), popularidad (por popularidad promedio, **popAvg**), aleatorio (**rnd**), bandidos (ϵ-greedy no personalizado, basado en objetos, **bandit**). Para la opción de vecinos próximos, factorización de matrices, popularidad y aleatorio, se han utilizado las implementaciones de RankSys[1]. Para la opción de bandidos, se ha utilizado las implementaciones realizadas en kNN-Bandit[2].

Para poder comparar de manera adecuada con nuestro algoritmo de Deep Q-Learning,

[1] Saúl Vargas, Pablo Castells. Ranksys framework. http://ranksys.org, 2018
[2] Javier Sanz Cruzado, Pablo Castells. `https://github.com/ir-uam/kNNBandit`, 2019

necesitamos ejecutar estos algoritmos teniendo en cuenta cómo ejecutábamos DDQN. En este, cada episodio correspondía a un solo usuario, al que le recomendábamos 40 objetos. Es por ello, que para cada estrategia de recomendación se han generado 40 recomendaciones para cada usuario. Posteriormente, se recorre cada usuario y se va contabilizando la relevancia descubierta. Para ello, con cada rating predicho, se busca el par usuario-objeto en el conjunto de datos de test. Si está, y el rating era mayor que el umbral de relevancia, se suma uno a la cuenta de ratings relevantes descubiertos. Al final del proceso, dividimos por el número total de ratings relevantes presentes en el conjunto de datos de test. Así, podemos visualizar el porcentaje de relevancia que va siendo descubierta a través de los episodios.

En la Figura 4.6 se muestran los resultados de la comparativa. Lo primero que salta a la vista es el pobre rendimiento de las estrategias clásicas de recomendación: hkv, ub y popAvg. El motivo es la distribución de los usuarios en nuestro conjunto de datos. Como indicamos en el Apartado 4.1, hay 491 usuarios que no cuentan con puntuaciones en el conjunto de entrenamiento. Es por ello, que hay muchos usuarios para los que no se generan puntuaciones, o en su defecto, se generan 40 ceros, y por tanto, ralentizan muchísimo el descubrimiento de relevancia. Esto nos sugiere un escenario de arranque muy frío, donde no se conoce las preferencias de un gran porcentaje de usuarios, y por tanto, el experimento nos permite extraer una conclusión clara: DDQN es muy superior al resto de estrategias de recomendación cuando se cuenta con pocos datos de preferencias de los usuarios. Además, la alternativa de los bandits es claramente inferior durante todos los episodios, viéndose superado hasta por la estrategia aleatoria cuando nos acercamos a los 400 episodios. También cabe destacar que DDQN, en test, cuenta con un $\epsilon = 0{,}05$. Es decir, explora el 5 % de las veces. Es por ello que sigue descubriendo relevancia de forma lineal, aunque vemos que rnd lo supera en los últimos episodios. Sin embargo, ofrece una ventaja clara contra el resto de estrategias durante gran parte del proceso, y sobre todo en los primeros episodios, cuando es más importante acertar para no perder a los usuarios.

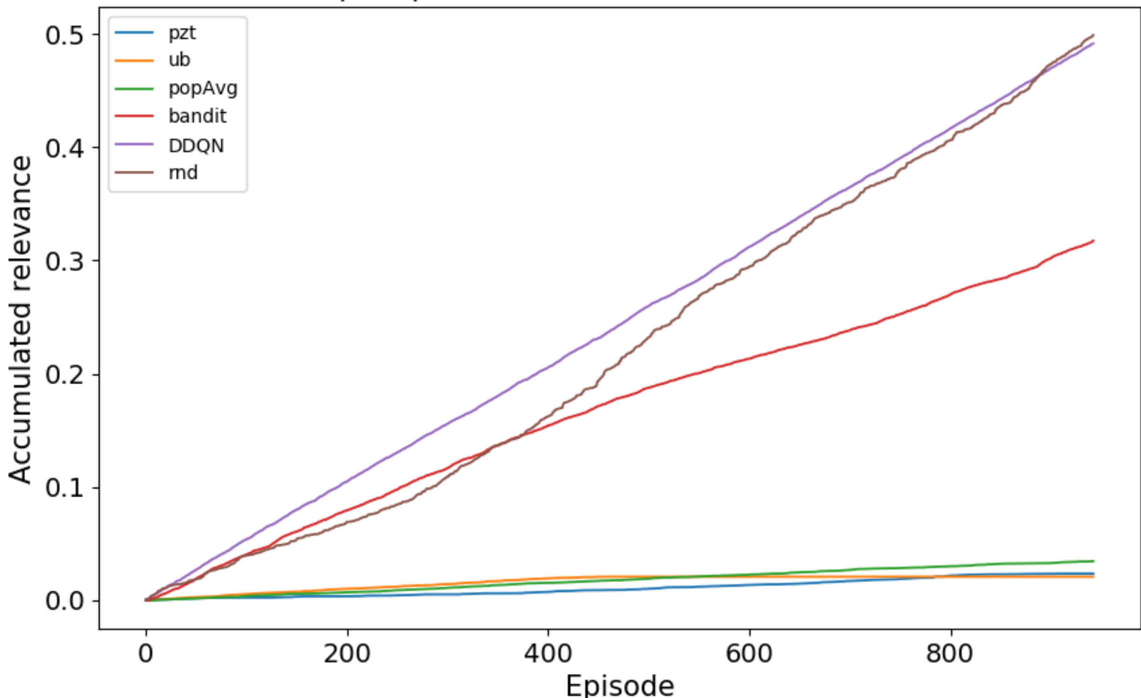

Figura 4.6: Relevancia descubierta para los diferentes algoritmos de recomendación propuestos.

5

Conclusiones y trabajo futuro

5.1. Conclusiones

Durante este trabajo, hemos realizado un estudio del campo del Reinforcement Learning, y hemos aplicado un método novedoso, presentado en [23], que combina las ideas de los Procesos de Decisión de Markov con la extracción de características latentes mediante Factorización de Matrices. Además, aunque el método ya había sido propuesto, se ha realizado un desarrollo intensivo implementando el sistema de simulación de recomendación recursiva, la gestión de los datos, la evaluación, todos los mecanismos propios del Deep Q-Learning utilizando la librería Keras-RL [1], etc.

A la vista de los resultados, podemos extraer varias conclusiones. La primera, y más general de todas, es que el campo del Reinforcement Learning puede ser aplicado al campo de la recomendación, conformando un área totalmente novedosa y prometedora para poder ser aplicada a problemas y sistemas de recomendación. La siguiente, es que dependiendo del modelado de los estados y acciones, y del algoritmo elegido para resolver el MDP, podemos explorar multitud de soluciones y comprobar si tienen buen rendimiento, bajo diferentes configuraciones y datasets.

Centrándonos más en la solución desarrollada en este trabajo, hemos visto que hay pocas diferencias entre las 3 alternativas de Deep Q-Learning propuestas: DQN, DDQN y D3QN. Esto se puede deber a que, al utilizar el vector de características como estado, gran parte de la información sobre las preferencias de los usuarios, y las relaciones entre estos, se codifican ya en estas características. Por ello, aunque utilicemos métodos más sofisticados de Deep Q-Learning, no obtenemos grandes mejoras de rendimiento. Además, estas pequeñas variaciones de rendimiento se pueden deber al factor aleatorio que introducimos gracias al uso de políticas ϵ-greedy, y que regulamos, como es obvio, con

el propio ϵ.

Además, a la hora de comparar estrategias de recomendación, escogiendo una selección que tenga representación de todas las áreas principales dentro del campo (vecinos próximos, factorización de matrices, bandits, etc.), observamos que la alternativa del Deep Q-Learning aporta una mejora muy clara. La situación vivida con el COVID-19 no ha permitido profundizar en estos experimentos, y tan solo se ha podido comprobar con un dataset, sesgado por popularidad, y con particiones de entrenamiento y test desequilibradas, (muchos usuarios no aparecen en entrenamiento), lo que puede simular una configuración de arranque muy frío, donde el sistema apenas tiene información inicial sobre las preferencias de los usuarios. Sin embargo, dentro de esta configuración, observamos como los métodos de Reinforcement Learning (bandidos, DDQN), mejoran claramente las estrategias clásicas de recomendación, por lo que una conclusión clara es que aplicar Reinforcement Learning en un contexto de arranque frío es una muy buena opción. Por último, no podemos dejar de señalar que dentro del campo, el Deep Q-Learning aporta mejores resultados que los bandidos.

5.2. Trabajo Futuro

Como hemos mencionado anteriormente, el desarrollo de experimentos más complejos y más completos queda como principal trabajo futuro. El uso de diferentes datasets, que aporten visiones complementarias, (usar algún dataset que no tenga sesgo por popularidad en las puntuaciones, datasets de otros ámbitos: música, redes sociales, etc.) es una de las ideas principales y más inmediatas que se podrían aplicar. Además, dentro de esos conjuntos de datos, el uso de diferentes particiones de entrenamiento y test puede aportarnos nuevas claves sobre el rendimiento de las técnicas de Reinforcement Learning, en comparación de algoritmos clásicos. En definitiva, cualquier complejidad que añadamos a los experimentos nos aportaría nueva información sobre si el campo aporta mejoras sobre los algoritmos desarrollados hasta ahora.

En lo referente a la solución propuesta, se podrían incluir modificaciones a la hora de modelar la tarea de recomendación como un MDP. Por ejemplo, podría incorporarse el procesamiento de los estados dentro de la propia Q-Network, usando una capa de Embedding. Esta idea viene fundamentada por el estudio realizado en [36], donde los autores argumentan que se puede usar un Word Embedding como un Factorizador de Matrices implícito. Así, obtendríamos un sistema que, primero, sería mucho más sencillo de entender cómo funciona, y que simplificaría bastante la arquitectura, integrándolo todo en una sola red neuronal. Además, se podrían realizar multitud de cambios sobre el propio agente, como utilizar una representación del estado diferente, o diseñar una recompensa más compleja. También podría modelarse la tarea de recomendación como un POMDP [37], utilizando estados y acciones ocultos.

Por último, se podría profundizar en la evaluación offline. Se podrían utilizar diferentes modelos de feedback del usuario, simulando que el usuario da feedback solo a los relevantes, solo al más relevante, entre otros. Además, podrían utilizarse diferentes métricas, que

representen no solo la cantidad de relevancia descubierta, sino también la precisión de las recomendaciones realizadas.

Bibliografía

[1] Matthias Plappert. *keras-rl*. https://github.com/keras-rl/keras-rl. 2016.

[2] F.O. Isinkaye, Y.O. Folajimi y B.A. Ojokoh. «Recommendation systems: Principles, methods and evaluation». En: *Egyptian Informatics Journal* 16.3 (2015), págs. 261 -273. ISSN: 1110-8665. DOI: https://doi.org/10.1016/j.eij.2015.06.005. URL: http://www.sciencedirect.com/science/article/pii/S1110866515000341.

[3] Jure Leskovec, Anand Rajaraman y Jeffrey David Ullman. *Mining of Massive Datasets*. 2nd. New York, NY, USA: Cambridge University Press, 2014. ISBN: 1107077230, 9781107077232.

[4] Mehdi Elahi, Francesco Ricci y Neil Rubens. «A Survey of Active Learning in Collaborative Filtering Recommender Systems». En: *Comput. Sci. Rev.* 20.C (mayo de 2016), págs. 29-50. ISSN: 1574-0137. DOI: 10.1016/j.cosrev.2016.05.002. URL: http://dx.doi.org/10.1016/j.cosrev.2016.05.002.

[5] Neil Rubens y col. «Active Learning in Recommender Systems». En: *Recommender Systems Handbook*. Ed. por Francesco Ricci, Lior Rokach y Bracha Shapira. Boston, MA: Springer US, 2015, págs. 809-846. DOI: 10.1007/978-1-4899-7637-6_24. URL: https://doi.org/10.1007/978-1-4899-7637-6_24.

[6] Christian Desrosiers y George Karypis. «A Comprehensive Survey of Neighborhood-based Recommendation Methods». En: *Recommender Systems Handbook*. Ed. por Francesco Ricci y col. Boston, MA: Springer US, 2011, págs. 107-144. ISBN: 978-0-387-85820-3. DOI: 10.1007/978-0-387-85820-3_4. URL: https://doi.org/10.1007/978-0-387-85820-3_4.

[7] Andrew I. Schein y col. «Methods and Metrics for Cold-start Recommendations». En: *Proceedings of the 25th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. SIGIR '02. Tampere, Finland: ACM, 2002, págs. 253-260. ISBN: 1-58113-561-0. DOI: 10.1145/564376.564421. URL: http://doi.acm.org/10.1145/564376.564421.

[8] Guy Shani y Asela Gunawardana. «Evaluating Recommendation Systems». En: *Recommender Systems Handbook*. Ed. por Francesco Ricci y col. Boston, MA: Springer US, 2011, págs. 257-297. ISBN: 978-0-387-85820-3. DOI: 10.1007/978-0-387-85820-3_8. URL: https://doi.org/10.1007/978-0-387-85820-3_8.

[9] R. Sabbatini. *Neurons and synapses: the history of its dicovery*. 2003. URL: http://www.cerebromente.org.br/n17/history/neurons3_i.htm.

[10] Warren S. McCulloch y Walter Pitts. «A Logical Calculus of the Ideas Immanent in Nervous Activity». En: *Neurocomputing: Foundations of Research*. Cambridge, MA, USA: MIT Press, 1988, 15–27. ISBN: 0262010976.

[11] J. C. Cuevas-Tello. *Apuntes de Redes Neuronales Artificiales*. 2018. arXiv: 1806.05298 [cs.NE].

[12] David E. Rumelhart, Geoffrey E. Hinton y Ronald J. Williams. «Learning Representations by Back-Propagating Errors». En: *Neurocomputing: Foundations of Research*. Cambridge, MA, USA: MIT Press, 1988, 696–699. ISBN: 0262010976.

[13] Vinod Nair y Geoffrey E. Hinton. «Rectified Linear Units Improve Restricted Boltzmann Machines». En: *Proceedings of the 27th International Conference on International Conference on Machine Learning*. ICML'10. Haifa, Israel: Omnipress, 2010, 807–814. ISBN: 9781605589077.

[14] Kaiming He y col. *Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification*. 2015. arXiv: 1502.01852 [cs.CV].

[15] Y. Bengio y X. Glorot. «Understanding the difficulty of training deep feed forward neural networks». En: *International Conference on Artificial Intelligence and Statistics* (ene. de 2010), págs. 249-256.

[16] Andrew Y. Ng. «Feature Selection, L1 vs. L2 Regularization, and Rotational Invariance». En: *Proceedings of the Twenty-First International Conference on Machine Learning*. ICML '04. Banff, Alberta, Canada: Association for Computing Machinery, 2004, pág. 78. ISBN: 1581138385. DOI: 10.1145/1015330.1015435. URL: https://doi.org/10.1145/1015330.1015435.

[17] Nitish Srivastava y col. «Dropout: A Simple Way to Prevent Neural Networks from Overfitting». En: *Journal of Machine Learning Research* 15 (jun. de 2014), págs. 1929-1958.

[18] Richard S. Sutton y Andrew G. Barto. *Reinforcement Learning: An Introduction*. Second. The MIT Press, 2018. URL: http://incompleteideas.net/book/the-book-2nd.html.

[19] F.L. Lewis y Draguna Vrabie. «Reinforcement Learning and Adaptive Dynamic Programming for Feedback Control». En: *Circuits and Systems Magazine, IEEE* 9 (ene. de 2009), págs. 32 -50. DOI: 10.1109/MCAS.2009.933854.

[20] Alessandro Lazaric, Marcello Restelli y Andrea Bonarini. «Reinforcement Learning in Continuous Action Spaces through Sequential Monte Carlo Methods». En: *Proceedings of the 20th International Conference on Neural Information Processing Systems*. NIPS'07. Vancouver, British Columbia, Canada: Curran Associates Inc., 2007, 833–840. ISBN: 9781605603520.

[21] Xiangyu Zhao y col. *Deep Reinforcement Learning for List-wise Recommendations*. 2017. arXiv: 1801.00209 [cs.LG].

[22] Lixin Zou y col. *Reinforcement Learning to Optimize Long-term User Engagement in Recommender Systems*. 2019. arXiv: 1902.05570 [cs.IR].

[23] Yu Lei y Wenjie Li. «Interactive Recommendation with User-Specific Deep Reinforcement Learning». En: *ACM Trans. Knowl. Discov. Data* 13.6 (oct. de 2019). ISSN: 1556-4681. DOI: 10.1145/3359554. URL: https://doi.org/10.1145/3359554.

[24] Hado van Hasselt, Arthur Guez y David Silver. *Deep Reinforcement Learning with Double Q-learning*. 2015. arXiv: 1509.06461 [cs.LG].

[25] Scott Fujimoto, Herke van Hoof y David Meger. *Addressing Function Approximation Error in Actor-Critic Methods*. 2018. arXiv: 1802.09477 [cs.AI].

[26] Ziyu Wang y col. *Dueling Network Architectures for Deep Reinforcement Learning*. 2015. arXiv: 1511.06581 [cs.LG].

[27] Olivier Jeunen. «Revisiting Offline Evaluation for Implicit-Feedback Recommender Systems». En: *Proceedings of the 13th ACM Conference on Recommender Systems*. RecSys '19. Copenhagen, Denmark: Association for Computing Machinery, 2019, 596–600. ISBN: 9781450362436. DOI: 10.1145/3298689.3347069. URL: https://doi.org/10.1145/3298689.3347069.

[28] Xiangyu Zhao y col. «Toward Simulating Environments in Reinforcement Learning Based Recommendations». En: *CoRR* abs/1906.11462 (2019). arXiv: 1906.11462. URL: http://arxiv.org/abs/1906.11462.

[29] Guanjie Zheng y col. «DRN: A Deep Reinforcement Learning Framework for News Recommendation». En: *Proceedings of the 2018 World Wide Web Conference*. WWW '18. Lyon, France: International World Wide Web Conferences Steering Committee, 2018, 167–176. ISBN: 9781450356398. DOI: 10.1145/3178876.3185994. URL: https://doi.org/10.1145/3178876.3185994.

[30] Xiangyu Zhao y col. «Deep Reinforcement Learning for Page-Wise Recommendations». En: *Proceedings of the 12th ACM Conference on Recommender Systems*. RecSys '18. Vancouver, British Columbia, Canada: Association for Computing Machinery, 2018, 95–103. ISBN: 9781450359016. DOI: 10.1145/3240323.3240374. URL: https://doi.org/10.1145/3240323.3240374.

[31] Hongzi Mao y col. «Resource Management with Deep Reinforcement Learning». En: nov. de 2016, págs. 50-56. DOI: 10.1145/3005745.3005750.

[32] Greg Brockman y col. *OpenAI Gym*. 2016. eprint: arXiv:1606.01540.

[33] Diederik P. Kingma y Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2014. arXiv: 1412.6980 [cs.LG].

[34] F. Maxwell Harper y Joseph A. Konstan. «The MovieLens Datasets: History and Context». En: *ACM Trans. Interact. Intell. Syst.* 5.4 (dic. de 2015). ISSN: 2160-6455. DOI: 10.1145/2827872. URL: https://doi.org/10.1145/2827872.

[35] István Pilászy, Dávid Zibriczky y Domonkos Tikk. «Fast Als-Based Matrix Factorization for Explicit and Implicit Feedback Datasets». En: *Proceedings of the Fourth ACM Conference on Recommender Systems*. RecSys '10. Barcelona, Spain: Association for Computing Machinery, 2010, 71–78. ISBN: 9781605589060. DOI: 10.1145/1864708.1864726. URL: https://doi.org/10.1145/1864708.1864726.

[36] Omer Levy y Yoav Goldberg. «Neural Word Embedding as Implicit Matrix Factorization». En: *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*. NIPS'14. Montreal, Canada: MIT Press, 2014, 2177–2185.

[37] Zhongqi Lu y Qiang Yang. *Partially Observable Markov Decision Process for Recommender Systems*. 2016. arXiv: `1608.07793 [cs.AI]`.

Apéndices

Código desarrollado

A.1. Definiendo el sistema de gestión de ratings

```
import numpy as np
import pickle


class Ratings:

    def __init__(self):
        """
        Initializes a Ratings object

        """
        # Dict with user as key, another dict as value,
        # with item as key and rating as value
        self.user_ratings = {}
        # Dict with item as key, another dict as value,
        # with user as key and rating as value
        self.item_ratings = {}
        # Number of ratings
        self.n_ratings = 0
        # Number of different users
        self.num_users = 0
        # Number of different items
        self.num_items = 0
```

```
        # Mean global rating
        self.mean_rating = 0
        # Np array of users
        self.users_array = np.array([], dtype=int)
        # Np array of items
        self.items_array = np.array([], dtype=int)
        # User with highest id
        self.highest_user = 0
        # Item with highest id
        self.highest_item = 0

    # Adds a rating to the object
    def add_rating(self, user, item, rating):

        if user > self.highest_user:
            self.highest_user = user

        if item > self.highest_item:
            self.highest_item = item

        if user not in self.users_array:
            self.users_array = np.append(self.users_array, user)
            self.user_ratings[user] = {}
            self.num_users += 1

        if item not in self.items_array:
            self.items_array = np.append(self.items_array, item)
            self.item_ratings[item] = {}
            self.num_items += 1

        self.user_ratings[user][item] = rating
        self.item_ratings[item][user] = rating
        self.n_ratings += 1

        self.mean_rating += rating
        if self.n_ratings > 1:
            self.mean_rating /= 2

    # Returns the rating of the user u and item i,
    # if present, 0 otherwise
    def rating(self, u, i):
        if u in self.user_ratings:
            if i in self.user_ratings[u]:
                return self.user_ratings[u][i]
        return 0
```

```
# Returns two lists, containing the user-items ratings
def rated_user_item_pairs(self):
    users = []
    items = []
    for u in self.user_ratings.keys():
        for i in self.user_ratings[u].keys():
            users.append(u)
            items.append(i)
    return users, items

# Returns the items to which user u has given a rating (np array)
def rated_items(self, u):
    return np.fromiter(self.user_ratings[u].keys(), dtype=int)

# Returns the users that have rated an item (np array)
def rated_users(self, i):
    return np.fromiter(self.item_ratings[i].keys(), dtype=int)

# Returns the number of ratings that user u has in the dataset
def n_rated_items(self, u):
    return len(list(self.user_ratings[u].keys()))

# Returns the items that have a rating for the users u and v
def item_intersection(self, u, v):
    u_items = list(self.user_ratings[u].keys())
    v_items = list(self.user_ratings[v].keys())
    return [item for item in u_items if item in v_items]

# Returns a random user
def sample_user(self):
    return np.random.choice(self.users())

# Returns a random item that has a rating for that user
def sample_user_item(self, user):
    items = list(self.user_ratings[user].keys())
    return items[np.random.randint(len(items))]

# Returns a np array of items
def items(self):
    return self.items_array

# Returns a np array of users
def users(self):
    return self.users_array

# Return the number of items
```

```
    def n_items(self):
        return self.num_items

    # Returns the number of users
    def n_users(self):
        return self.num_users

    # Returns a user item rating matrix
    # (rows user, columns item, values rating)
    def ratings_matrix(self):
        matrix = np.zeros([self.highest_user+1, self.highest_item+1])
        for u in self.user_ratings.keys():
            for i in self.user_ratings[u].keys():
                matrix[u][i] = self.user_ratings[u][i]
        return matrix

    # Returns the number of ratings that are above a relevance threshold
    def n_relevant_ratings(self, relevance_threshold):
        n_ratings = 0
        for u in self.user_ratings.keys():
            for i in self.user_ratings[u].keys():
                if self.user_ratings[u][i] >= relevance_threshold:
                    n_ratings += 1
        return n_ratings

    # Saves the entire object using pickle module
    def save(self, filename):
        with open(filename, 'wb') as output:
            pickle.dump(self, output, pickle.HIGHEST_PROTOCOL)

    # Computes the std deviation of the ratings
    def std_deviation(self):
        ratings = []
        for u in self.user_ratings.keys():
            for i in self.user_ratings[u].keys():
                ratings.append(self.user_ratings[u][i])
        return np.std(np.array(ratings))


def read_ratings_split(train_file,
                       test_file,
                       user_index,
                       item_index,
                       sep="\t",
                       skip_first_line=False):
    """
```

```
    Reads a ratings split, where there are 4 files: one for
    train split, another for test split, and another 2 with user
    and item indexes.

    :param train_file: path to the training ratings file.
    :param test_file: path to the testing ratings file.
    :param user_index: path to the user index file.
    :param item_index: path to the item index file.
    :param sep: separator used in all the files.
    :param skip_first_line: boolean indicating if the first line
    should be skipped in all the files.
    :return: 2 ratings object, (train, test) filled with data
    present in the files.
    """
train = Ratings()
test = Ratings()

with open(user_index) as fp:
    users = fp.readlines()

with open(item_index) as fp:
    items = fp.readlines()

with open(train_file) as fp:

    if skip_first_line:
        fp.readline()

    line = fp.readline()
    c = 1
    while line:
        tokens = line.split(sep)
        user = int(tokens[0])
        item = int(tokens[1])
        rating = float(tokens[2])
        train.add_rating(user, item, rating)
        line = fp.readline()
        c += 1

with open(test_file) as fp:

    if skip_first_line:
        fp.readline()

    line = fp.readline()
    c = 1
```

```python
        while line:
            tokens = line.split(sep)
            user = int(tokens[0])
            item = int(tokens[1])
            rating = float(tokens[2])
            test.add_rating(user, item, rating)
            line = fp.readline()
            c += 1

    return train, test, users, items


def split_train_test_user_based(ratings_file,
                                train_size,
                                sep="\t",
                                skip_first_line=False):
    """
        Splits a single ratings file in two Ratings objects, one for
        train ratings, and another for test ratings. Each user belongs
        entirely into one of the two partitions. In another way,
        one user cannot have ratings in both train and test.

        :param ratings_file: file where every line is a tuple
        user-item-rating.
        :param train_size: percentage of train users
        (a number between 0 and 1).
        :param sep: separator of the ratings file
        :return: two Ratings objects, first the train one,
        and then the test one.
        """

    train = Ratings()
    test = Ratings()
    train_users = []
    test_users = []

    with open(ratings_file) as fp:

        if skip_first_line:
            fp.readline()

        line = fp.readline()
        while line:
            tokens = line.split(sep)
            user = int(tokens[0])
            item = int(tokens[1])
```

```
            rating = float(tokens[2])

            if user in train_users:
                train.add_rating(user, item, rating)
            elif user in test_users:
                test.add_rating(user, item, rating)
            else:
                if np.random.uniform(0, 1) <= train_size:
                    train.add_rating(user, item, rating)
                    train_users.append(user)
                else:
                    test.add_rating(user, item, rating)
                    test_users.append(user)

            line = fp.readline()

    return train, test
```

A.2. Definiendo las políticas y el entorno

```
import gym
import numpy as np
from spaces import Array, LatentVector
from rl.policy import Policy, LinearAnnealedPolicy


class ResettableLinearAnnealedPolicy(LinearAnnealedPolicy):
    def reset(self, attr=None, value=None):
        if attr is None:
            self.inner_policy.reset()
        else:
            setattr(self.inner_policy, attr, value)

    def reset_action_set(self, new_actions):
        self.inner_policy.reset_action_set(new_actions)


class OneTimeEpsGreedyPolicy(Policy):

    def __init__(self, nb_actions, eps=0.1):
        """
        Initialize an epsilon greedy policy, where each action can
        only be selected once, until reset function is called.
```

```
        :param nb_actions: (int) number of actions
        :param eps: epsilon
        """
        super(OneTimeEpsGreedyPolicy, self).__init__()
        self.eps = eps
        self.actions = np.array(list(range(0, nb_actions)))
        self.possible_actions = np.copy(self.actions)

    def select_action(self, q_values):
        """Return the selected action

        # Arguments
            q_values (np.ndarray): List of the estimations of Q
            for each action

        # Returns
            Selection action
        """

        if self.possible_actions.shape[0] > 0:
            assert q_values.ndim == 1

            if np.random.uniform() < self.eps:
                idx = np.random.randint(self.possible_actions.shape[0])
                action = self.possible_actions[idx]
            else:
                max_q = float('-inf')
                action = 0
                i = 0
                for a in self.actions:
                    if a in self.possible_actions:
                        if q_values[a] > max_q:
                            max_q = q_values[a]
                            action = a
                            idx = i
                        i += 1

            self.possible_actions = np.delete(self.possible_actions, idx)
            return action
        else:
            """
            A reset of the policy could be done.
            """
            self.reset()
            return self.select_action(q_values)
```

```
    def reset(self):
        self.possible_actions = np.copy(self.actions)

    def reset_action_set(self, new_actions):
        self.actions = new_actions
        self.possible_actions = np.copy(self.actions)

    def get_config(self):
        """Return configurations of EpsGreedyPolicy

        # Returns
            Dict of config
        """
        config = super(OneTimeEpsGreedyPolicy, self).get_config()
        config['eps'] = self.eps
        return config


class LatentStateEnv(gym.Env):
    """
        Gym environment that uses a latent features vector as state
    """
    metadata = {'render.modes': ['human']}

    def __init__(self, train, user, mf_model, max_episode_steps=40):
        """
        Initializes the object.

        :param train: ratings object
        :param user: integer of the user
        :param mf_model: udqn_mf object
        :param max_episode_steps: int, number of steps per episode
        """
        self.mf_model = mf_model
        self.user = user
        self.train = train
        # Initialize action and observation space
        self.action_space = Array(train.items())
        self.observation_space = LatentVector(self.mf_model.n_factors)
        # Initialize raw and latent states
        self.latent_state = self.mf_model.user_vecs[user]
        self.raw_state = np.array([])
        self.max_episode_steps = max_episode_steps
        self.current_episode_steps = 0

    def step(self, action):
```

```
        # Check if the rating is present in the train data
        r = self.train.rating(self.user,
                              action)
        if r > 0:
            # If its present, updates the latent state
            self.raw_state = np.append(self.raw_state,
                                       action)
            self.latent_state = self.mf_model.latent_state(self.user,
                                                           action,
                                                           r)

        self.current_episode_steps += 1
        return (self.latent_state,
                r,
                self.current_episode_steps >= self.max_episode_steps,
                {})

    def reset(self):
        self.latent_state = self.mf_model.reset()
        self.raw_state = np.array([])
        self.current_episode_steps = 0
        return self.latent_state

    def render(self, mode='human', close=False):
        print("Raw state: {}".format(self.raw_state))
        # print("Latent state: {}".format(self.latent_state))
```

A.3. Definiendo el agente y la Q-Network

```
import keras
import tensorflow
from keras import Input, Model
from keras.layers import Dense, Flatten, Embedding
from keras.models import Sequential
from keras.optimizers import Adam
from rl.agents import DQNAgent
from rl.memory import SequentialMemory


def build_model(n_hidden_layers, n_hidden_neurons, n_inputs,
                n_outputs):
    """
    It constructs the NN model.
```

```
    :param n_hidden_layers: integer indicating the number of hidden
↪   layers of the NN.
    :param n_hidden_neurons: list in witch each i-th position indicates
↪   the number of neurons of the i-th hidden layer.
    :param n_inputs: number of input neurons.
    :param n_outputs: number of output neurons (n_actions).

    :return: tensorflow NN model
    """
    model = Sequential()

    model.add(Flatten(input_shape=(1, n_inputs)))

    counter = 0
    while counter < n_hidden_layers:
        model.add(
            Dense(n_hidden_neurons[counter], activation='sigmoid'))
        counter += 1

    model.add(Dense(n_outputs, activation='linear'))

    model.compile(loss='mse', optimizer='adam', metrics=['mae'])
    model.summary()
    return model


class QNetwork:
    def __init__(self,
                 model,
                 policy,
                 test_policy,
                 nb_actions,
                 dqn_mode,
                 gamma=0.2,
                 weights_file=None):
        self.model = model
        self.policy = policy
        self.test_policy = test_policy
        memory = SequentialMemory(limit=50000, window_length=1)

        enable_double_dqn = False
        enable_dueling_network = False
        dueling_type = None
        if dqn_mode == "t":
            enable_double_dqn = True
        elif dqn_mode == "d":
```

```
            enable_double_dqn = True
            enable_dueling_network = True
            dueling_type = "avg"

        self.dqn = DQNAgent(model=model,
                            nb_actions=nb_actions,
                            memory=memory,
                            nb_steps_warmup=100,
                            enable_double_dqn=enable_double_dqn,

                             ↪  enable_dueling_network=enable_dueling_network,
                            dueling_type=dueling_type,
                            gamma=gamma,
                            target_model_update=1e-2,
                            policy=policy,
                            test_policy=test_policy)

        if weights_file is not None:
            self.dqn.load_weights(weights_file)
        self.dqn.compile(Adam(lr=1e-3), metrics=['mae'])

    def train(self,
              env,
              nb_steps=50000,
              visualize=True,
              verbose=2,
              callbacks=None):
        return self.dqn.fit(env,
                            nb_steps,
                            visualize=visualize,
                            verbose=verbose,
                            callbacks=callbacks)

    def test(self,
             env,
             nb_episodes=1,
             visualize=False,
             verbose=2,
             callbacks=None,
             nb_max_episode_steps=1000):
        return self.dqn.test(env,
                             nb_episodes=nb_episodes,
                             visualize=visualize,
                             verbose=verbose,
                             callbacks=callbacks,
                             nb_max_episode_steps=nb_max_episode_steps)
```

```
    def save(self, filename):
        self.dqn.save_weights(filename, overwrite=True)

    def load(self, filename):
        self.dqn.load_weights(filename)
```

A.4. Entrenando el agente

```
import sys
import gym
import latent_state_env
import locale
import numpy as np
from latent_state_env.envs.latent_state_env import \
    OneTimeEpsGreedyPolicy, ResettableLinearAnnealedPolicy

from src.mf.online_mf import OnlineMF
from src.qnetwork.qnetwork import build_model, QNetwork
from src.ratings.ratings_util import read_ratings_split
from src.util import save_object

import matplotlib.pyplot as plt

"""
    Main configuration.
    N_ARGUMENTS: int, number of input arguments of the script
"""
N_ARGUMENTS = 4

"""
    MF Configuration
    N_LATENT_FACTORS: int, number of dimensions of the latent vectors.
    MF_ITERATIONS: int, number of iterations to compute the latent
                   vectors.
    PRECOMPUTE_ITERATIONS: int, number of iterations to precompute
                           the model.
"""
N_LATENT_FACTORS = 16
MF_ITERATIONS = 20
MF_LEARNING_RATE = 0.1
PRECOMPUTE_ITERATIONS = 20

"""
```

```
    QNetwork Configuration
    N_HIDDEN_LAYERS: int, number of hidden layers of the
                          neural network.
    N_HIDDEN_NEURONS: list, with size=N_HIDDEN_LAYERS, in which
                           element is an int, indicating the number
                           of neurons of the corresponding hidden
                           layer.
    N_INPUTS: number of neurons of the input layer.
"""
N_HIDDEN_LAYERS = 2
N_HIDDEN_NEURONS = [64, 64]
N_INPUTS = N_LATENT_FACTORS

"""
    Agent Configuration
    GAMMA: float, discount factor.
    EPSILON: float, epsilon-greedy parameter.
    VERBOSE: boolean, debug flag.
    VISUALIZE: boolean, debug flag.
    MAX_EPISODE_STEPS_TRAIN: int, number of max steps per episode.
"""
GAMMA = 0.4
EPSILON = 0.1
VERBOSE = True
VISUALIZE = False
MAX_EPISODE_STEPS_TRAIN = 40

if len(sys.argv[1:]) < N_ARGUMENTS:
    print(
        "Call the program like this:\npython main dataset{ml-100k, "
        "cm-100k, Webscope_R3} partition{20-80, 80-20} "
        "rel_threshold mode{n nb}")
    sys.exit()
else:
    dataset = sys.argv[1:][0]
    split = sys.argv[1:][1]
    rel_threshold = int(sys.argv[1:][2])

    mode = sys.argv[1:][3]
    if len(sys.argv[1:]) > 4:
        # "t" for target network (double qlearning),
        # "d" for dueling qlearning
        dqn_mode = sys.argv[1:][4]
        if dqn_mode == "t":
            dqn_name = "double"
        else:
```

```
        dqn_name = "dueling"
else:
    dqn_mode = ""
    dqn_name = "base"

TRAIN_PATH = "../datasets/" + split + "/" + dataset + "/train" + "_"
↪   + mode + ".txt"
TEST_PATH = "../datasets/" + split + "/" + dataset + "/test" + "_" +
↪   mode + ".txt"
USER_INDEX_PATH = "../datasets/" + split + "/" + dataset + "/uIndex"
↪   + "_" + mode + ".txt"
ITEM_INDEX_PATH = "../datasets/" + split + "/" + dataset + "/iIndex"
↪   + "_" + mode + ".txt"
QNETWORK_FILEPATH = "../objects/qnetwork_" + str(N_HIDDEN_NEURONS[
                                                     0]) +
                                                     ↪  "_neurons_"
                                                     ↪  + split +
                                                     ↪  "_" +
                                                     ↪  dataset +
                                                     ↪  "_" + mode +
                                                     ↪  "_" +
                                                     ↪  dqn_name +
                                                     ↪  "_" + str(
    GAMMA) + ".txt"
MF_MODEL_FILEPATH = "../objects/mf_model_" + str(N_HIDDEN_NEURONS[
                                                     0]) +
                                                     ↪  "_neurons_"
                                                     ↪  + split +
                                                     ↪  "_" +
                                                     ↪  dataset +
                                                     ↪  "_" + mode +
                                                     ↪  "_" +
                                                     ↪  dqn_name +
                                                     ↪  "_" + str(
    GAMMA) + ".txt"
HISTORY_FILEPATH = "../objects/history_train_" + str(
    N_HIDDEN_NEURONS[
        0]) + "_neurons_" + split + "_" + dataset + "_" + mode + "_"
        ↪  + dqn_name + "_" + str(
    GAMMA) + ".txt"
FIGURE_FILEPATH = "../images/train_reward_" + str(
    N_HIDDEN_NEURONS[
        0]) + "_neurons_" + dataset + "_" + split + "_" + mode + "_"
        ↪  + dqn_name + "_" + str(
    GAMMA) + ".png "
```

```
    ACC_REWARD_FIGURE_FILEPATH = "../images/train_accumulated_reward_" +
    ↪   str(
        N_HIDDEN_NEURONS[
            0]) + "_neurons_" + dataset + "_" + split + "_" + mode + "_"
            ↪   + dqn_name + "_" + str(
        GAMMA) + ".png"

locale.setlocale(locale.LC_ALL, 'es_ES')

"""
Firstly, we read the split already made in the datasets folder.
"""
train, test, users, items = read_ratings_split(TRAIN_PATH,
                                               TEST_PATH,
                                               USER_INDEX_PATH,
                                               ITEM_INDEX_PATH)
n_users = len(users)
n_items = len(items)
train_users = np.copy(train.users())

"""
Initialize the keras model of the QNetwork, and the Qnetwork itself.
The number of inputs is the number of latent factors, the number of
outputs is the number of items, and the configuration of the network
is given in configuration time (needs to be predefined).
"""
n_outputs = n_items

keras_model = build_model(N_HIDDEN_LAYERS,
                          N_HIDDEN_NEURONS,
                          N_INPUTS,
                          n_outputs)

aux_policy = OneTimeEpsGreedyPolicy(nb_actions=n_items,
                                    eps=EPSILON)

policy = ResettableLinearAnnealedPolicy(aux_policy,
                                        attr='eps',
                                        value_max=1.0,
                                        value_min=EPSILON,
                                        value_test=0.05,
                                        nb_steps=10000)

test_policy = OneTimeEpsGreedyPolicy(nb_actions=n_items,
                                     eps=EPSILON)
```

```
qnetwork = QNetwork(keras_model,
                    policy,
                    test_policy,
                    nb_actions=n_items,
                    gamma=GAMMA,
                    dqn_mode=dqn_mode)

"""
Initialize the matrix factorisation model. We instantiate it,
and precompute it, in order to get the initial item latent vectors.
"""
mf_model = OnlineMF(train.user_ratings,
                    n_users,
                    n_items,
                    N_LATENT_FACTORS,
                    MF_LEARNING_RATE)
mf_model.precompute(n_iter=PRECOMPUTE_ITERATIONS)

"""
Train the QNetwork.
"""
np.random.shuffle(train_users)
# Each episode is a single user
n_train_users = train_users.shape[0]
n_users_trained = 0
history = {'episode_reward': [], 'nb_steps': [],
           'accumulated_reward': []}
print_counter = 0
episode = 0
accumulated_reward = 0
for u in train_users:
    if VERBOSE:
        if print_counter >= 40:
            n_users_trained += 40
            print("{} users trained of {} in total".format(
                n_users_trained, len(train_users)))
            print_counter = 0

    policy.reset_action_set(train.rated_items(u))
    env = gym.make('latentstate-v0',
                   train=train,
                   user=u,
                   mf_model=mf_model,
                   max_episode_steps=MAX_EPISODE_STEPS_TRAIN)
    history_keras = qnetwork.train(env,
                                   nb_steps=MAX_EPISODE_STEPS_TRAIN,
```

```
                                   verbose=VERBOSE,
                                   visualize=VISUALIZE)
    accumulated_reward += history_keras.history['episode_reward'][0]
    history['episode_reward'].append(
        history_keras.history['episode_reward'][0])
    history['nb_steps'].append(
        history_keras.history['nb_steps'][0] * (episode + 1))
    history['accumulated_reward'].append(accumulated_reward)
    episode += 1
    print_counter += 1

qnetwork.save(QNETWORK_FILEPATH)
save_object(mf_model, MF_MODEL_FILEPATH)
save_object(history, HISTORY_FILEPATH)

plt.plot(range(0, episode), history['episode_reward'], label="Reward")
plt.title(
    "Training Reward per Episode of {} ({})".format(dataset, split))
plt.legend()
if int(len(train_users) / 10) > 0:
    plt.xticks(np.arange(0, episode, int(len(train_users) / 10)))
else:
    plt.xticks(range(0, episode))
# plt.show()
plt.savefig(FIGURE_FILEPATH)
plt.clf()

plt.plot(range(0, episode),
         history['accumulated_reward'],
         label="Accumulated Reward")
plt.title("Training Accumulated Reward per "
          "Episode of {} ({})".format(dataset, split))
plt.legend()
if int(len(train_users) / 10) > 0:
    plt.xticks(np.arange(0, episode, int(len(train_users) / 10)))
else:
    plt.xticks(range(0, episode))
# plt.show()
plt.savefig(ACC_REWARD_FIGURE_FILEPATH)
plt.clf()
```

A.5. Evaluando el agente

```
import sys
from statistics import mean

import gym
import latent_state_env
import locale
import numpy as np
from latent_state_env.envs.latent_state_env import
↪  OneTimeEpsGreedyPolicy, ResettableLinearAnnealedPolicy

from src.qnetwork.qnetwork import build_model, QNetwork
from src.ratings.ratings_util import read_ratings_split
from src.util import load_object, save_object

import matplotlib.pyplot as plt

"""
    Main configuration.
    N_ARGUMENTS: int, number of input arguments of the script
"""
N_ARGUMENTS = 4

"""
    MF Configuration
    N_LATENT_FACTORS: int, number of dimensions of the latent vectors.
    MF_ITERATIONS: int, number of iterations to compute the latent
                    vectors.
    PRECOMPUTE_ITERATIONS: int, number of iterations to precompute
                            the model.
"""
N_LATENT_FACTORS = 16
MF_ITERATIONS = 20
MF_LEARNING_RATE = 0.1
PRECOMPUTE_ITERATIONS = 20

"""
    QNetwork Configuration
    N_HIDDEN_LAYERS: int, number of hidden layers of the
                      neural network.
    N_HIDDEN_NEURONS: list, with size=N_HIDDEN_LAYERS, in which
                       element is an int, indicating the number
                       of neurons of the corresponding hidden
                       layer.
```

```
    N_INPUTS: number of neurons of the input layer.
"""
N_HIDDEN_LAYERS = 2
N_HIDDEN_NEURONS = [64, 64]
N_INPUTS = N_LATENT_FACTORS

"""
    Agent Configuration
    GAMMA: float, discount factor.
    EPSILON: float, epsilon-greedy parameter.
    VERBOSE: boolean, debug flag.
    VISUALIZE: boolean, debug flag.
    MAX_EPISODE_STEPS: int, number of max steps per episode.
"""
GAMMA = 0.4
EPSILON = 0.1
VERBOSE = True
VISUALIZE = False
MAX_EPISODE_STEPS = 40


if len(sys.argv[1:]) < N_ARGUMENTS:
    """
    Call the program like this:
    python main dataset partition
    """
    print(
        "Call the program like this:\npython main_test dataset{"
        "ml-100k, cm-100k, Webscope_R3} partition{20-80, "
        "80-20} rel_threshold mode{n nb}")
    sys.exit()
else:
    dataset = sys.argv[1:][0]
    split = sys.argv[1:][1]
    rel_threshold = int(sys.argv[1:][2])

    mode = sys.argv[1:][3]

    if len(sys.argv[1:]) > 4:
        # "t" for target network (double qlearning),
        # "d" for dueling qlearning
        dqn_mode = sys.argv[1:][4]
        if dqn_mode == "t":
            dqn_name = "double"
        else:
            dqn_name = "dueling"
```

```
    else:
        dqn_mode = ""
        dqn_name = "base"

    TRAIN_PATH = "../datasets/" + split + "/" + dataset + "/train" + "_"
     ↪  + mode + ".txt"
    TEST_PATH = "../datasets/" + split + "/" + dataset + "/test" + "_" +
     ↪  mode + ".txt"
    USER_INDEX_PATH = "../datasets/" + split + "/" + dataset + "/uIndex"
     ↪  + "_" + mode + ".txt"
    ITEM_INDEX_PATH = "../datasets/" + split + "/" + dataset + "/iIndex"
     ↪  + "_" + mode + ".txt"
    TEST_RESULTS_PATH = "../results/" + split + "/" + dataset + "/" +
     ↪  dataset + "_" + mode + "_rl.txt"
    QNETWORK_FILEPATH = "../objects/qnetwork_" + str(N_HIDDEN_NEURONS[0])
     ↪  + "_neurons_" \
                        + split + "_" + dataset + "_" + mode + "_" +
                         ↪  dqn_name + "_" + str(GAMMA) + ".txt"
    MF_MODEL_FILEPATH = "../objects/mf_model_" + str(N_HIDDEN_NEURONS[0])
     ↪  + "_neurons_" \
                        + split + "_" + dataset + "_" + mode + "_" +
                         ↪  dqn_name + "_" + str(GAMMA) + ".txt"
    QNETWORK_TEST_FILEPATH = "../objects/qnetwork_test_" + split + "_" +
     ↪  dataset + "_" + mode + "_" + dqn_name + "_" + str(GAMMA) + ".txt"
    MF_MODEL_TEST_FILEPATH = "../objects/mf_model_test_" + split + "_" +
     ↪  dataset + "_" + mode + "_" + dqn_name + "_" + str(GAMMA) + ".txt"
    EPISODE_REWARDS_FILEPATH = "../objects/history_test_" + split + "_" +
     ↪  dataset + "_" + mode + "_" + dqn_name + "_" + str(GAMMA) + ".txt"
    EPISODE_RECALLS_FILEPATH = "../objects/recall_test_" + split + "_" +
     ↪  dataset + "_" + mode + "_" + dqn_name + "_" + str(GAMMA) + ".txt"
    FIGURE_REWARD_FILEPATH = "../images/" + dataset + "_" + split + "_" +
     ↪  mode + "_" + dqn_name + "_" + str(GAMMA) + "_test_reward.png"
    FIGURE_RECALL_FILEPATH = "../images/" + dataset + "_" + split + "_" +
     ↪  mode + "_" + dqn_name + "_" + str(GAMMA) + "_test_recall.png"

locale.setlocale(locale.LC_ALL, 'es_ES')

"""
Firstly, we read the split already made in the datasets folder.
"""
train, test, users, items = read_ratings_split(TRAIN_PATH,
                                               TEST_PATH,
                                               USER_INDEX_PATH,
                                               ITEM_INDEX_PATH)
n_users = len(users)
n_items = len(items)
```

```
train_users = np.copy(train.users())

"""
Initialize the keras model of the QNetwork, and the Qnetwork itself.
The number of inputs is the number of latent factors, the number of
outputs is the number of items, and the configuration of the network
is given in configuration time (needs to be predefined).
"""
n_outputs = n_items
keras_model = build_model(N_HIDDEN_LAYERS,
                          N_HIDDEN_NEURONS,
                          N_INPUTS,
                          n_outputs)

aux_policy = OneTimeEpsGreedyPolicy(nb_actions=n_items,
                                    eps=EPSILON)
policy = ResettableLinearAnnealedPolicy(aux_policy,
                                        attr='eps',
                                        value_max=1.0,
                                        value_min=EPSILON,
                                        value_test=0.05,
                                        nb_steps=10000)
test_policy = OneTimeEpsGreedyPolicy(nb_actions=n_items,
                                     eps=EPSILON)
qnetwork = QNetwork(keras_model,
                    policy,
                    test_policy,
                    dqn_mode=dqn_mode,
                    nb_actions=n_items,
                    gamma=GAMMA)
qnetwork.load(QNETWORK_FILEPATH)

mf_model = load_object(MF_MODEL_FILEPATH)

"""
Test the QNetwork.
"""
with open(TEST_RESULTS_PATH, "a") as fp:
    fp.write("")

test_users = np.copy(test.users())
np.random.shuffle(test_users)
episode_rewards = []
episode_recalls = []
n_test_users = test_users.shape[0]
total_relevant_ratings = test.n_relevant_ratings(rel_threshold)
```

```
discovered_relevant_ratings = 0
n_users_tested = 0
for u in test_users:
    if VERBOSE:
        print("Testing user {}. ({}/{})".format(u, n_users_tested,
         ↪  n_test_users), file=sys.stderr)
    test_policy.reset_action_set(test.rated_items(u))
    env = gym.make('latentstate-v0',
                   train=test,
                   user=u,
                   mf_model=mf_model,
                   max_episode_steps=MAX_EPISODE_STEPS)

    history = qnetwork.test(env,
                            nb_max_episode_steps=MAX_EPISODE_STEPS,
                            visualize=VISUALIZE,
                            verbose=VERBOSE)
    episode_rewards.append(history.history['episode_reward'][0])
    discovered_relevant_ratings += episode_rewards[-1]
    episode_recalls.append(discovered_relevant_ratings /
     ↪  total_relevant_ratings)
    n_users_tested += 1

qnetwork.save(QNETWORK_TEST_FILEPATH)
save_object(mf_model, MF_MODEL_TEST_FILEPATH)
save_object(episode_rewards, EPISODE_REWARDS_FILEPATH)
save_object(episode_recalls, EPISODE_RECALLS_FILEPATH)

plt.plot(range(0, len(episode_rewards)),
         episode_rewards,
         label="Reward")
plt.title("Testing Reward per Episode of {}, "
          "with a {} partition".format(dataset, split))
plt.legend()
plt.xticks(np.arange(0, len(episode_rewards),
                     int(len(test_users) / 10)))
# plt.show()
plt.savefig(FIGURE_REWARD_FILEPATH)
plt.clf()

plt.plot(range(0, len(episode_recalls)),
         episode_recalls,
         label="Recall")
plt.title("Testing Recall per Episode of {}, "
          "with a {} partition".format(dataset, split))
plt.legend()
```

```
plt.xticks(np.arange(0, len(episode_recalls),
                     int(len(test_users) / 10)))
# plt.show()
plt.savefig(FIGURE_RECALL_FILEPATH)
print("Final mean reward: {}".format(mean(episode_rewards)))
```